ALGONQUIN COLLEGE

SCHOOL OF ADVANCED TECHNOLOGY

ICT - Applications & Programming
Computer Engineering Technology – Computing Science

Compilers

LECTURE NOTES[1] / THEORY

This material is exclusive for CST8152 - Compilers Course at Algonquin College.

[1] Originally developed by **Prof. Svillen Ranev** (Algonquin College, 2020).

Welcome to a beautiful season in the Canadian Capital City…

Source: https://www.forbes.com/sites/miriamporter/2019/09/29/outdoor-fall-activities-in-ottawa

Lecture Notes – Fall, 2023

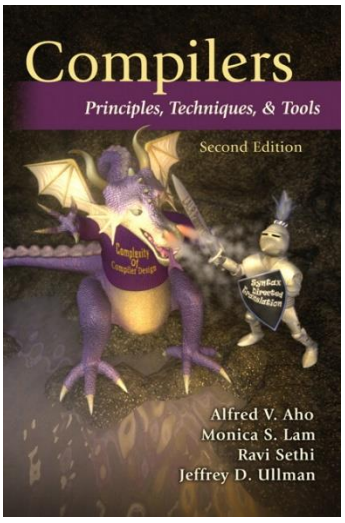
Preface

1

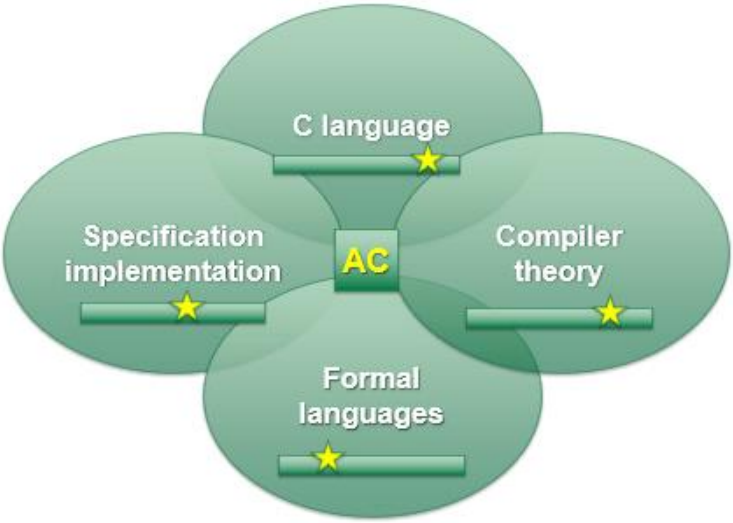
About this Document

Some initial notes:

- The articles summarize concepts, but they do not intend to replace the book. Complete details can be found at: **Compilers – Principles, Techniques & Tools**, 2nd ed., by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Published by Pearson (Addison Wesley).



- This course combines a set of concepts that will be integrated in the theoretical and practical components:



  - **Formal language**: when you need to define models and representations for one specific language.
  - **Compiler theory**: where concepts related to the construction of languages are discussed.
  - **Specific implementation**: that helps to define elements for one specific language implementation using proper definitions.
  - **C language**: The way you need to implement your language by using C programming features (including types, structs, unions, pointers).
- The practical component (Labs) is integrant part of the course and an important element of the final grade.
  - Compilers is not only a theoretical course, maybe the labs (and Assignments) are the most interesting part of it.
  - This time, students can choose to implement a DSL (Domain-Specific-Language) or a Rust derivation (see http://rust-lang.org/).

Articles covered in this version:

PART I – Compiler Basic

- **Article 1**: Introduction to Compilers
- **Article 2**: The Context of a Compiler
- **Article 3**: Inside Compilers
- **Article 4**: Compiler Development
- **Article 5**: Formal Languages
- **Article 6**: Grammars and Languages
- **Article 7**: Regular Expressions
- **Article 8**: Finite Automata
- **Article 9**: Transformations: RE to NFA
- **Article 10**: Transformations: NFA into DFA
- **Article 11**: Grammar Notations
- **Article 12**: Symbol Tables
- **Article 13**: Parsing Overview
- **Article 14**: Grammar and Parsing
- **Article 15**: Non-Recursive Predictive Parsing
- **Article 16**: Error Handling

PART II – Annexes

- **Annex 1**: Useful tables (ASCII, Unicode, Hexa)
- **Annex 2**: Useful C Language Concepts
- **Annex 3**: Complete ANSI C Reference Card
- **Annex 4**: Useful links

PART III – Language Specification

- **Lang 1**: Go Language Specification (Grammar / Examples)
- **Lang 2**: Sofia Language Specification (Informal / Formal)
- **Lang 3**: More Languages (additional language specs and examples)

Good luck with the course…

Compiler Professors,
Algonquin College,
Fall, 2023.

Part I

Lecture Notes

Article 1

Compilers: Introduction

1.1. Introduction

A **Compiler** (the term was coined by Grace Murrey Hopper in early 50s) is a program that runs on some computer architecture under some operating system and transforms (translates) an input program (source program) written in some programming language into an output program (target program) expressed in different programming language.

FIG 1.1 - Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

A **Programming Language** is a notational system for describing computations in machine-readable and human-readable form.

- Programming language compilers are part of a more general category of language manipulation tools or programs – Language Processors.
- Language processors include natural language translators and interpreters, text editors, text-to-speech and speech to text coverers, spell and grammar checkers and some others language tools.
- Programming language interpreters are subset of programming language compilers.

Computation in general is any process that can be carried by a computer. Programming Languages must provide two types of abstractions: **data abstractions** and **control abstractions**.

1.2. Importance: Why study compilers

Compiler construction is a very specialized computer science and system programming field – there are relatively limited number of programmers involved in writing compilers. So why should a computer science student spend time to learn exactly how a compiler is designed and written? There are three

main reasons:

FIG 1.2 - Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

- **Compilers are used by all programmers**. While most programmers will not ever write a full-fledged compiler, they use one every day. Knowledge about how compilers do their work can be used to write more efficient and error free code in a high-level language. As they say, "*A good craftsman should know his tools*." This is one of the most compelling to study compilers.

- **Compilers elements and techniques are used in almost every application**. Many programmers use compiler components for programming other applications. There is a good chance that a programmer will need to write a compiler or interpreter for a domain-specific language. Writing a parser for XML, HTML, or some other structured data file is a common task. Scanning and parsing a command or user input line is a very common task. Looking for a specific word or sentence in a text is a very common task.

- **Compilers are an excellent "capstone" or "focal" programming project.** Writing a compiler requires an understanding of almost all of the basic computer science subfields. To write a compiler, a programmer needs to know regular expressions, grammars, finite automata theory, programming paradigms, operating systems, computer architecture, a large range of data structures and algorithms, some programming languages, and a good and sound software engineering principles.

- **Finally**, it is considered a topic that you should know to be "well-educated" or "well-versed" in computer science.

1.3. Brief History:

Here is a basic list of programming languages that were developed and have some relationships.

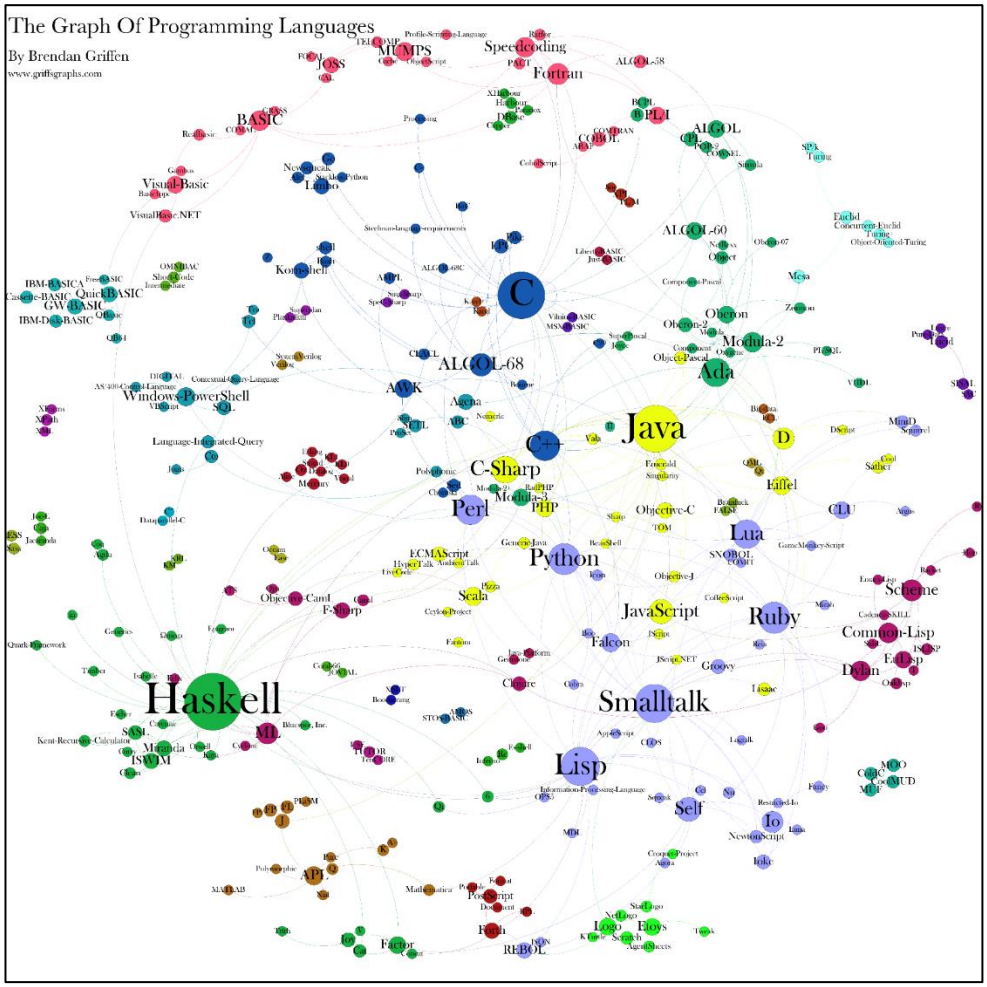


FIG 1.3 - Source: https://griffsgraphs.wordpress.com/2012/07/01/programming-languages-influences/

- **1830-40** – Analytical Engine invented by Charles Babbage. His wife Ada Augusta, Countess Lovelace (daughter of Lord Byron) wrote the first programs;
- **1950** – FORTRAN (Formula Translation System: 1954-1957), COBOL (COmmon Business Oriented Language: 1959-1960). Algol60, LISP (LISt Processor);
- **1960** – PL/1, SNOBOL (StriNg Oriented symbolic Language), Simula, BASIC, Logo;
- **1970** – Pascal (1971), C (1972);
- **1980** - Ada, Modula, Smalltalk-80 (1972-1980), C++ (1980-1985), Objective C, Object Pascal, Eiffel, Oberon, Scheme;
- **1990** – Java, Haskell, JavaScript, PHP, Perl, Python, Ruby, Lua;
- **2000** - C#, Scala , F#, Groovy, Go, D, R, Clojure, Swift, Kotlin;
- **2010** – Rust, Julia;
- **2020** – GPT-3, …

- **2021-22** – OPT, Platypus[2] / Sofia / Boa / Julius / Sofia[3] (I'm kidding) 😊…
- **2023** – ChatGPT (see https://openai.com/blog/chatgpt/).

Note that some sites (ex: **TIOBE** - https://www.tiobe.com/tiobe-index/) can also describe the visual representation of programming language evolution.

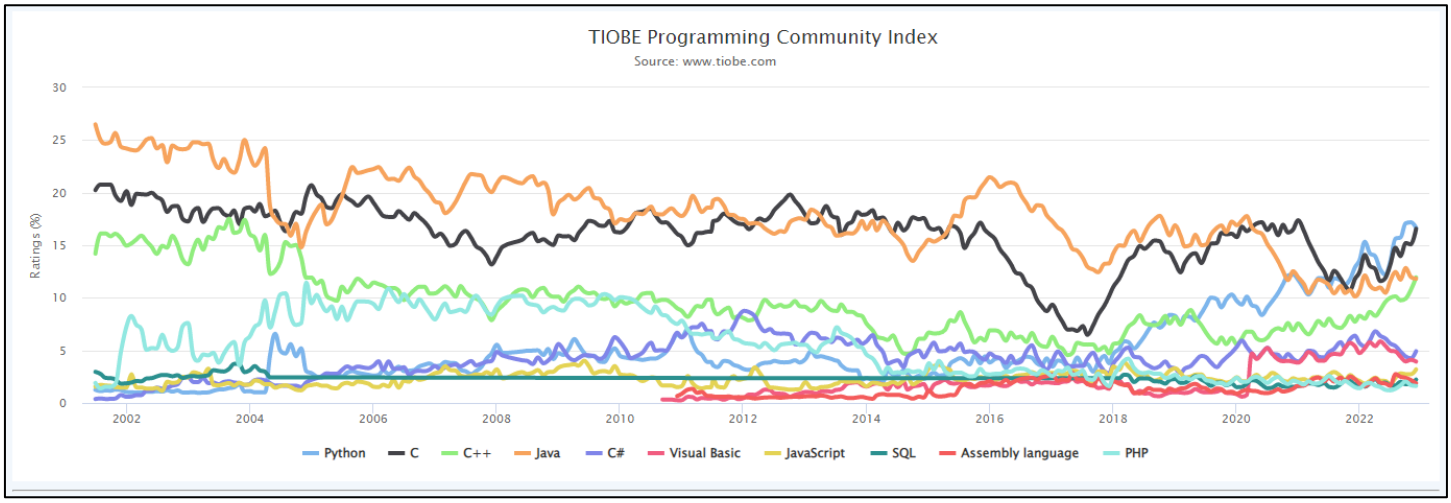


FIG 1.4 - Source: https://www.tiobe.com/tiobe-index/

Note: Rust evolution.

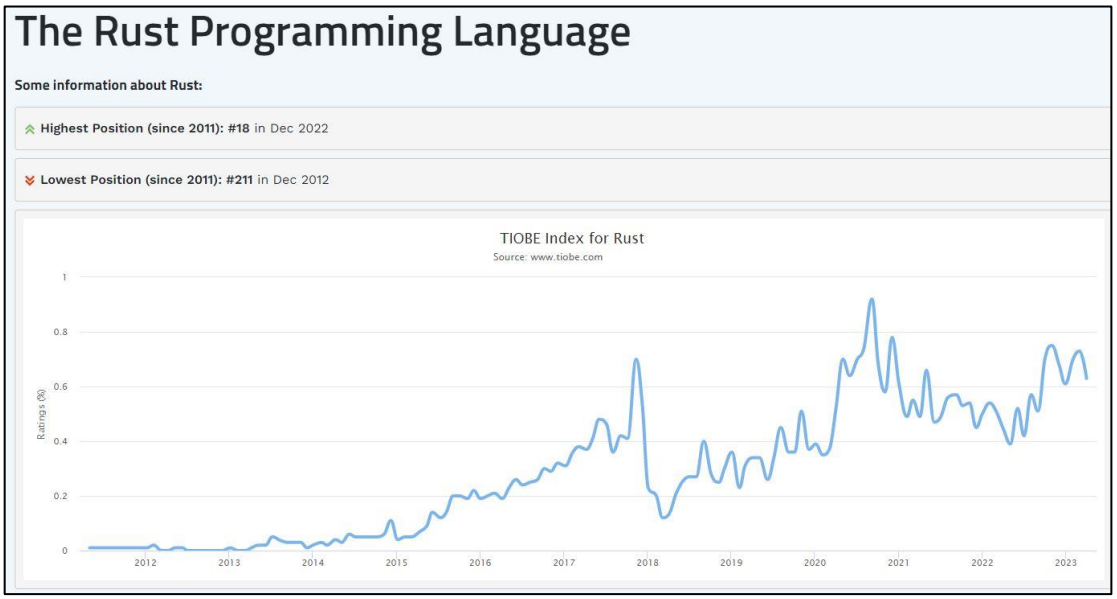


FIG 1.5 - Source: https://www.tiobe.com/tiobe-index/rust/

[2] The language created by prof. Svillen Ranev – **PLATYPUS** (acronym for **Programming Language – Aforethought Tiny Yet Procedural, Unascetic and Sophisticated**).

[3] The modification of PLATYPUS – the language "**MOLD**" has some similarity with **Rust** (rust-lang.org) as a language to be used during this term.

1.4. Concepts: general-purpose and special purpose.

General-purpose

- Computational Paradigms:
  - **Imperative** or **Procedural Programming** (based on John von Neumann's computer architecture). Examples:

    FORTRAN, COBOL, ALGOL, BASIC, PL, SNOBOL, C, ADA, Modula.
  - **Functional Programming** – based on functions as data values and lambda calculus.
    - Examples: LISP, Scheme, ML (Meta Language), Miranda, Haskell, F#, Clojure.
  - **Logic Programming** – based on symbolic logic or first-order predicate calculus. A logic programming language is a notational system for writing logical statements together with specified algorithms for implementing inference rules.
    - Examples Prolog
  - **Object-Oriented Programming** – based on the concept of an object, which can be described as a collection of memory locations together with all the operations (methods) that can change the value of these memory locations.
    - Examples: SIMULA, Smaltalk, C++, Objective C, Modula-3, Oberon, Eiffel, Object Pascal, JAVA, C#.
  - **Parallel Programming Languages** – allow for concurrent execution of computational processes.
    - Example: ADA, MPL and PVM libraries.
  - **Scripting Languages** – combine utilities, libraries, operating system commands into a program.
    - Examples**:** Perl, Python, Tcl/Tk, Javascript, Rexx, Visual Basic, PHP.
  - **Markup Languages –** define notations for specific objects, using proper semantics by conventions.
    - Example: SGML, HTML and XML.

Domain Specific Language (DSL) / Special-purpose Languages

- **Definition**: Languages define for proper utilizations. Here are some examples:
  - **Database Query Languages** – SQL

- **Simulation Languages** – Simula, GPSS, SIMSCRIPT
- **Silicon Design Languages** – VRML, VHDL,SystemC (C++), SpecC(C)
- **Graphics Design Languages** – GRAF
- **Real-time Languages** – RT-FORTRAN, BCL, Embedded-C, Embedded Java

1.5. Some Definitions:

Compiler Input:

- Source program
- Configuration parameters or pragmatics (#pragma directives)
- *Source and Target Language Definitions*

Compiler Output:

- Target program
- Error messages
- Information accompanying the target program – external symbol tables, cross- reference tables.

Example:

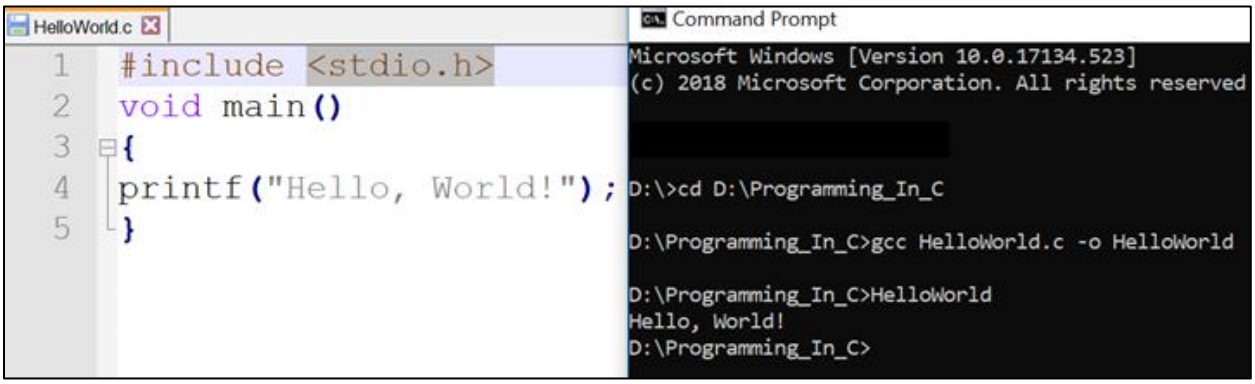


FIG 1.6 – C Code example

About targets:

- **Target Program:**
  - High-Level Language
  - Low-Level Code (Language)
- **Target Low-Level Code Type:**
  - Pure Machine Code
  - **Augmented Machine Code**

  - Virtual Machine Code
- **Target Low-Level Code Format:**
  - Assembly or Pseudo-assembly Language Format,
  - **Relocatable Binary Format**
  - Memory-Image Format (Load & Go)
- **Run-time Environment:**
  - Fully Static Environment
  - Fully Dynamic Environment
  - Mixed Environment – Stacked-based environment

1.6. Where to find compilers

Compiler Related Applications

- Editors, Word Processors, Command Interpreters, Formatting Printers, XML Parser, and almost all applications – big and small.

Review Questions

1. What is the importance of compilers?
2. Summarize the functionality of a compiler.
3. Identify some challenges to create a compiler.
4. What you believe about the future in compilers technology?

.

Article 2 The Context of a Compiler

2.1. Schematic View

See the process of compilation:

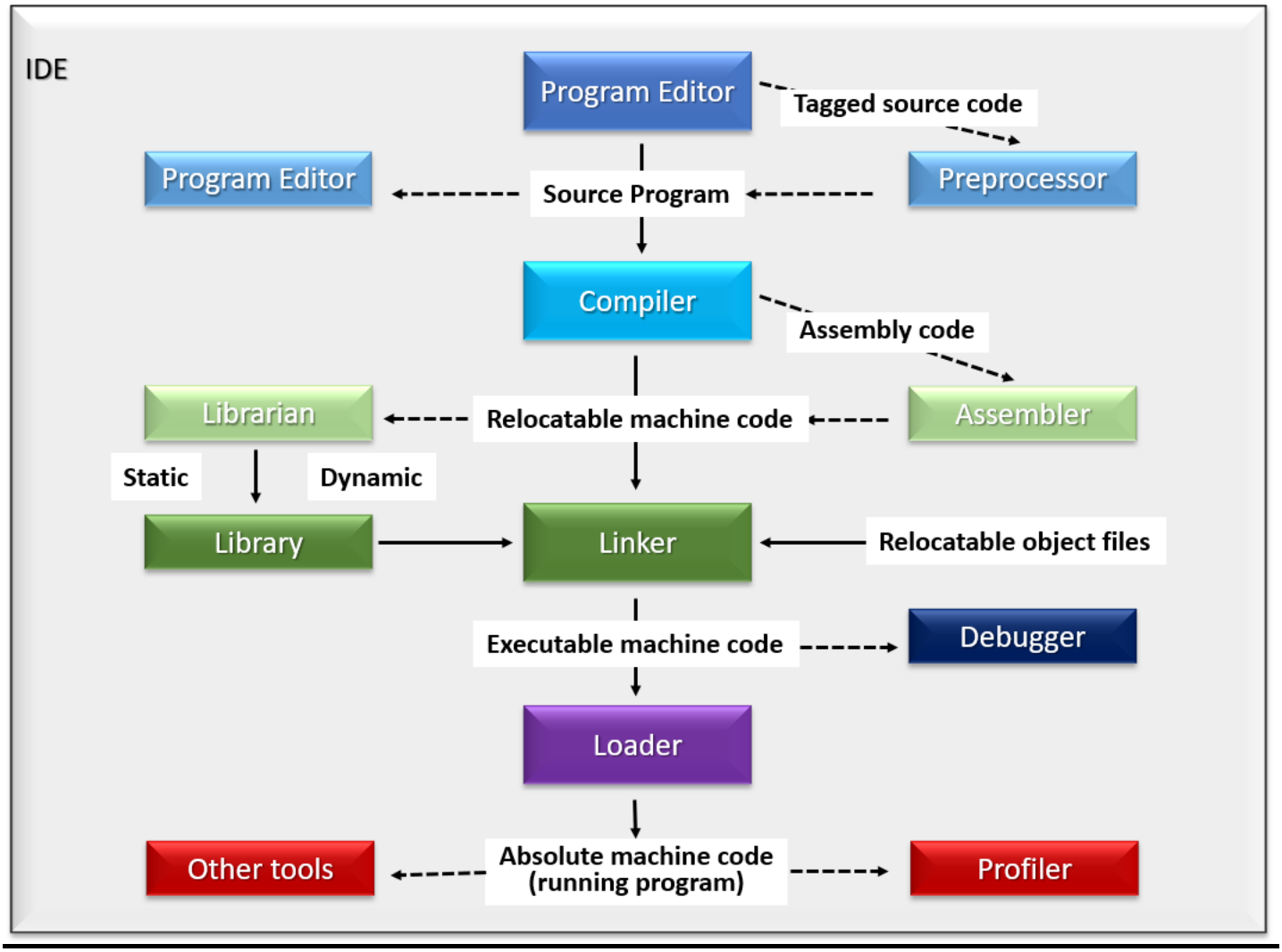
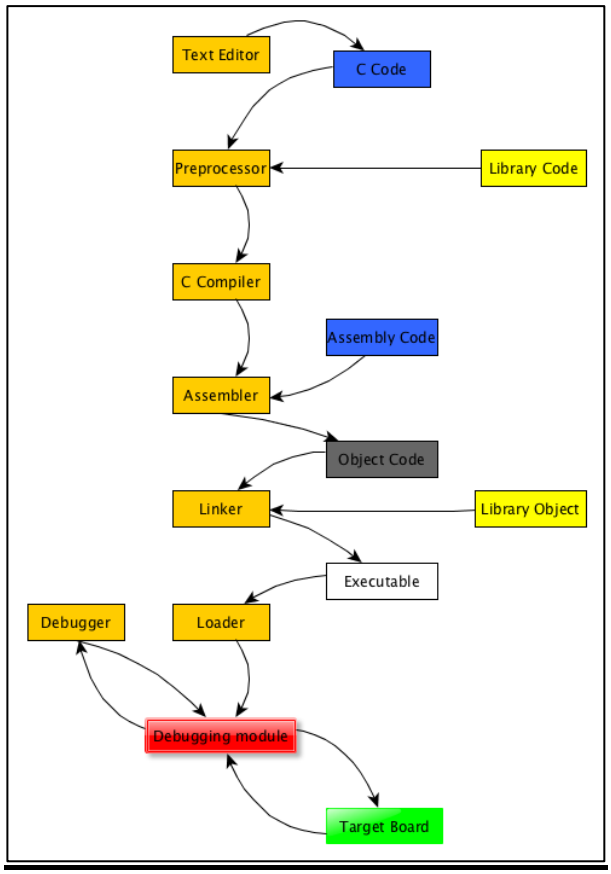


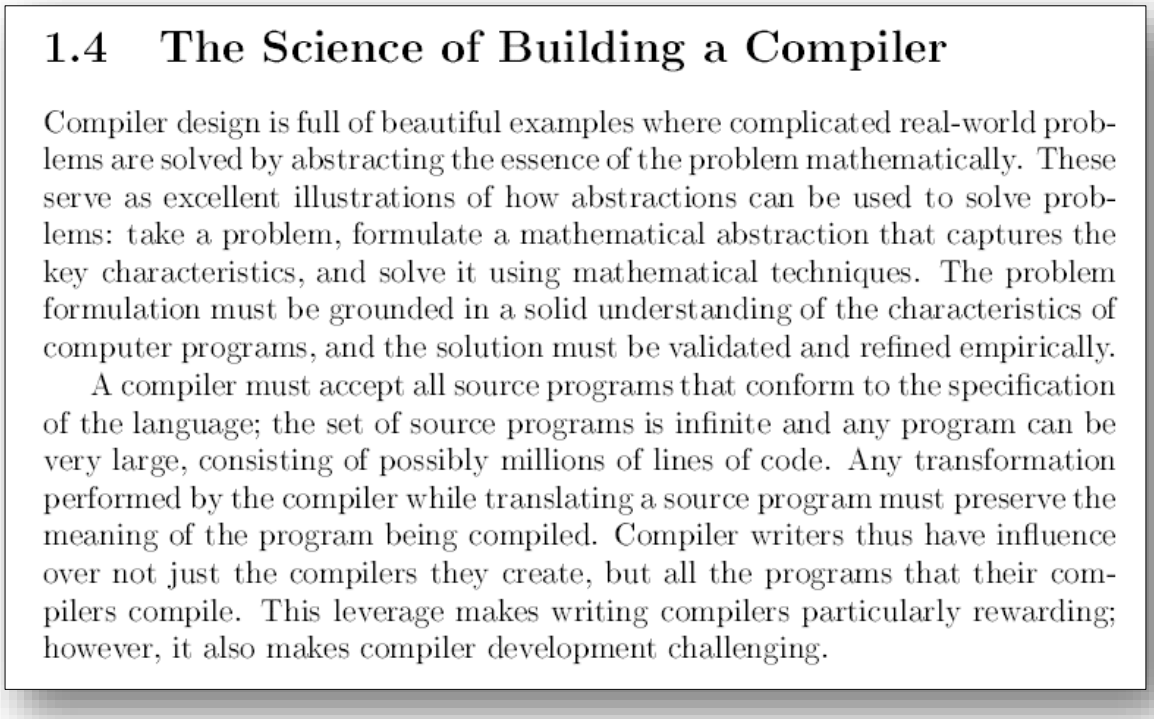
FIG 2.1 – Compilation tools

2.1. Definitions (alphabetic list)



- **Assembler**: Assemblers are simple compilers which translate assembly language into machine code.
- **Compiler**: Translate the text of the program in another language - usually assembler or some form of machine code.
- **Debugger**: Allows the user to trace the execution of a program statement by statement and inspect the content of different parts of the program memory.
- **Librarian**: Allows creating and maintaining libraries of pre-compiled component which can be used later without the need to be compiled again.
- **Linker**: Combines (links) all necessary components of a program into some executable form. Not all programming languages require linkers.
- **Loader**: Loads an executable program and passes the control to the program.
- **Preprocessor**: The purpose of the preprocessor is to augment automatically the code of the program following some directives provided by the programmer as a script. Preprocessors are language specific and not all programming languages have a preprocessor.
- **Program editor:** Allows the user to enter the text (code) of the program and save it in a text form (ASCII or Unicode). Any text editor can be used as program editors, but modern specialized program editors provide many additional features specific to the programming language. Modern program editor also incorporates some compiler elements like static syntax checking.

2.2. Other tools



1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

FIG 2.2 - Source: Aho (main reference)

- **Examples**:
    - Automatic or Unit testers (**Junit** - https://junit.org/junit5/),
    - Code Inspectors and Analyzers (https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers),
    - Error loggers (**Log4J** - https://logging.apache.org/log4j/2.x/),
    - Make and build scripting tools (**Ant** - https://ant.apache.org/),
    - Profilers (**JRat** - http://jrat.sourceforge.net/),
    - Project Managers (**Maven** - https://maven.apache.org/)
    - Refactoring tools (https://www.codeguru.com/tools/using-the-visual-studio-code-refactoring-tools/),
    - Run-time Inspectors (http://www.programmers-friend.org/JOI/),
    - Style Formatters (https://marketplace.visualstudio.com/search?target=VSCode&category=Formatters),
    - Task Managers (**Mylyn** - https://www.eclipse.org/mylyn/),
    - Version Control and, Collaboration (**Git** - https://git-scm.com/).

Review Questions

1. Describe the compilation process using the previous concepts.
2. Give some errors that can happen in this process.
3. Propose a strategy to identify and fix (when possible) these errors.

Article 3

Inside Compilers

3.1. General View

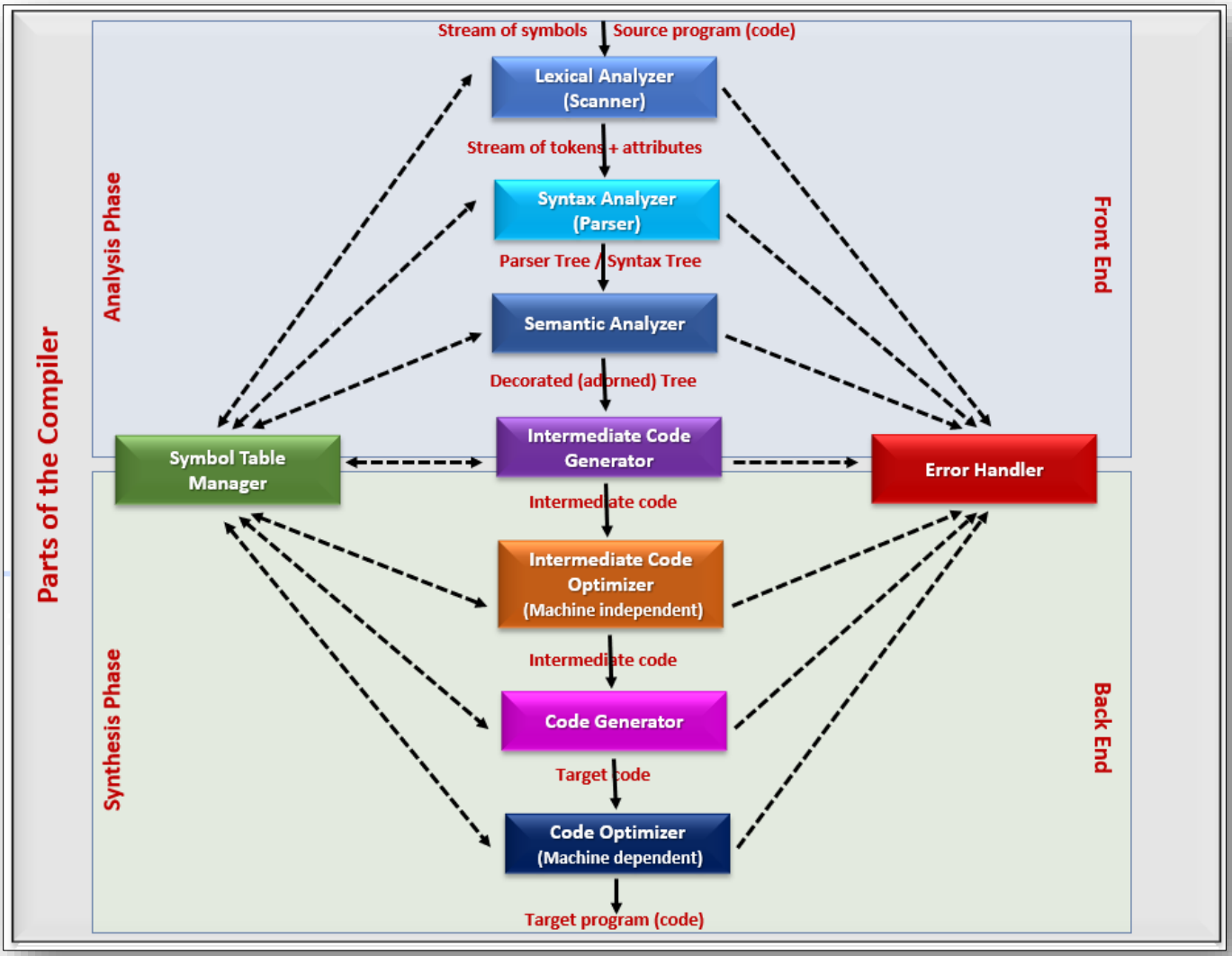


FIG 3.1 – Compilation process and phases

- The ***analysis part*** (***phase***) breaks up the source program into constituent pieces and imposes a grammatical structure on them.
    - It then uses this structure to create an intermediate representation of the source

program.

- The ***synthesis part*** (***phase***) constructs the desired target program from the intermediate representation and the information in the symbol table.

Note: The analysis part is often called the ***front end*** of the compiler; the synthesis part is the ***back end***.

- The first phase of a compiler is called ***lexical analysis*** or ***scanning***.
  - The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
  - For each lexeme, the lexical analyzer produces as output a token of the form: (token-name (code), attribute-value).

- The second phase of the compiler is ***syntax analysis*** or ***parsing***.
  - The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation called *parse tree* that depicts the grammatical structure of the token stream.
  - Typically, the parse tree is reduced to another form of representation called *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.

- The ***semantic analyzer*** uses the syntax tree and the information in the ***symbol table*** to check the source program for semantic consistency with the language definition.
  - It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

In the process of translating a source program into target code, compilers construct one or more intermediate representations, which can have a variety of forms. For example, syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or ***machine-like intermediate representation***, which we can think of as a program for an *abstract machine*.
  - This intermediate representation should have two important properties: it should be easy to produce, and it should be easy to translate into the target machine.
  - Typically, the intermediate representation separates the front end from the back end.

- The ***machine-independent code-optimization*** phase attempts to improve the intermediate code so that better target code will result.

- There is a great variation in the amount of ***code optimization*** different compilers perform.

In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

- The ***code generator*** takes as input an intermediate representation of the source program and maps it into the target language

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

- The ***symbol table*** is a data structure containing a record for each variable name, with fields for the attributes of the name.

- The ***error handler*** is responsible to report to the programmer the lexical, syntactical, and the semantic error discovered during the compilation process. It is also responsible to prevent the compiler from producing a target code is an error has been detected.

The discussion of ***phases*** deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a ***pass*** that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a specific target machine.

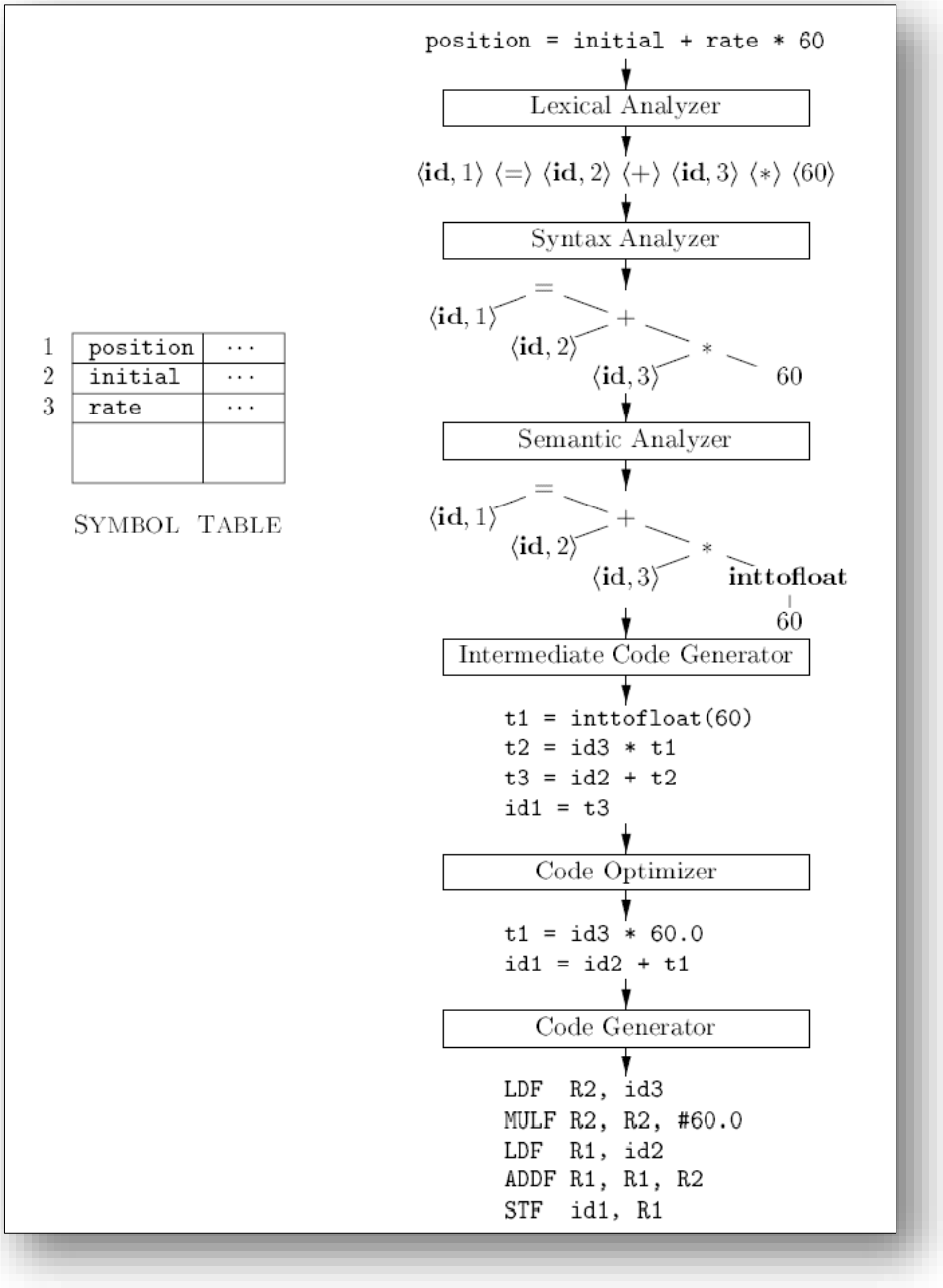


FIG 3.2: Language translation - Source: Aho (main reference)

The correct representation for each step is described in the following board:

1. `position` is a lexeme that would be mapped into a token ⟨**id**, 1⟩, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol `=` is a lexeme that is mapped into the token ⟨=⟩. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. `initial` is a lexeme that is mapped into the token ⟨**id**, 2⟩, where 2 points to the symbol-table entry for `initial`.
4. `+` is a lexeme that is mapped into the token ⟨+⟩.
5. `rate` is a lexeme that is mapped into the token ⟨**id**, 3⟩, where 3 points to the symbol-table entry for `rate`.
6. `*` is a lexeme that is mapped into the token ⟨∗⟩.
7. 60 is a lexeme that is mapped into the token ⟨60⟩.[1]

FIG 3.3: Token definitions - Source: Aho (main reference)

3.2. Some components of Compiler

- **Reviewing front-end compiler…**

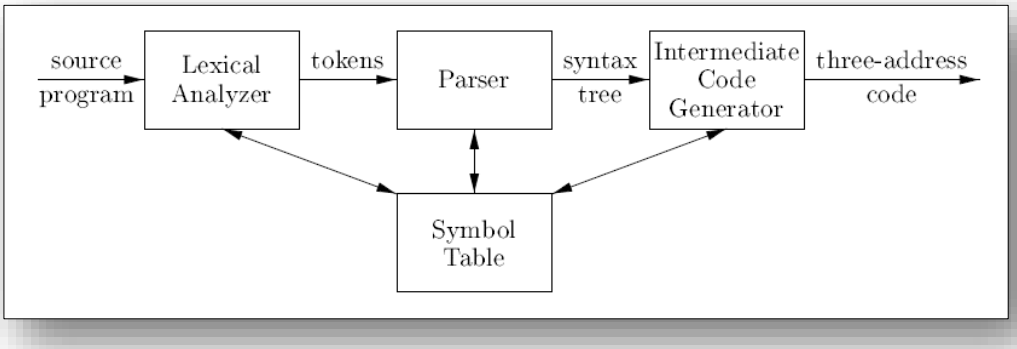


FIG 3.4: Schematic view - Source: Aho (main reference)

The buffer process…

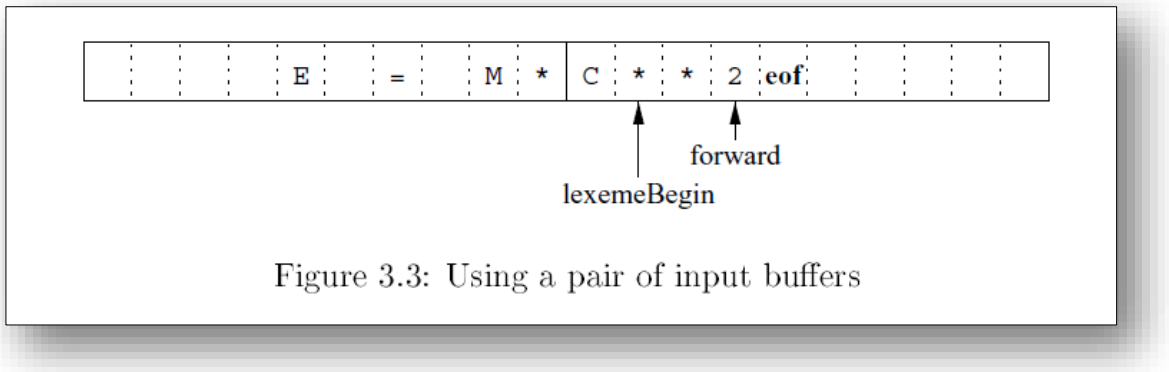


Figure 3.3: Using a pair of input buffers

FIG 3.5: Buffer utilization - Source: Aho (main reference)

Note that the buffer uses:

- Sentinels (to define lexemeBegin and forward);
- End of file (or final marks for buffer).

However, the buffer must be in a part of the memory and we need to manage the architecture.

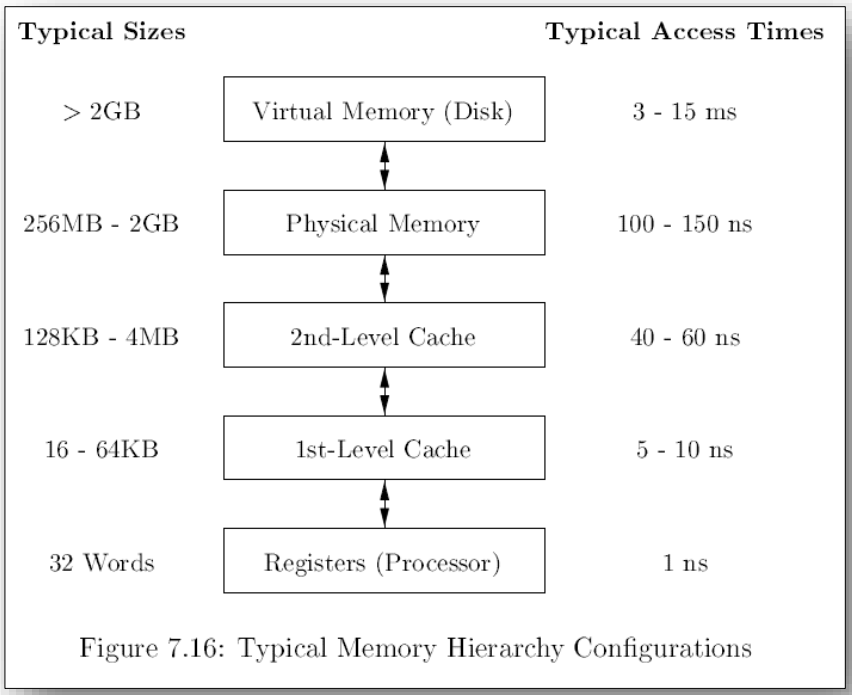


Figure 7.16: Typical Memory Hierarchy Configurations

FIG 3.6: Memory hierarchy - Source: Aho (main reference)

Review Questions

1. Explain the compilation process for a layman.
2. Exemplify the compilation for a specific language (for instance, Java or Python).
3. Give an idea about how to implement a language.

"*Why is that when a programmer writes code it is democracy but when a programmer runs a compiler it is a dictatorship?*" (Confused Philosopher).

Article 4 Compiler Development

4.1. Passes

- **Definition:** The sequence of steps followed during the compilation process. A **pass** reads the source and performs transformations, by intermediate files. Some languages require at least two passes to generate code.
- So, basically, we have:
    - **Single pass compiler**: The source code is scanned only once, done line by line. It is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
        - When the line source is processed, it is scanned and the token is extracted.
        - Then the syntax of each line is analyzed and the tree structure is built. After the semantic part, the code is generated.
        - The same process is repeated for each line of code until the entire program is compiled.
    - **Multi-pass compiler**: This is the common case, where we need to process the code several times and takes the result from one phase to another. We have independent machine code and splits the program into smaller pieces.
        - In the **first pass**, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
        - In the **second pass**, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
        - In the **third pass**, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax. This pass is going on, until the target output is produced.

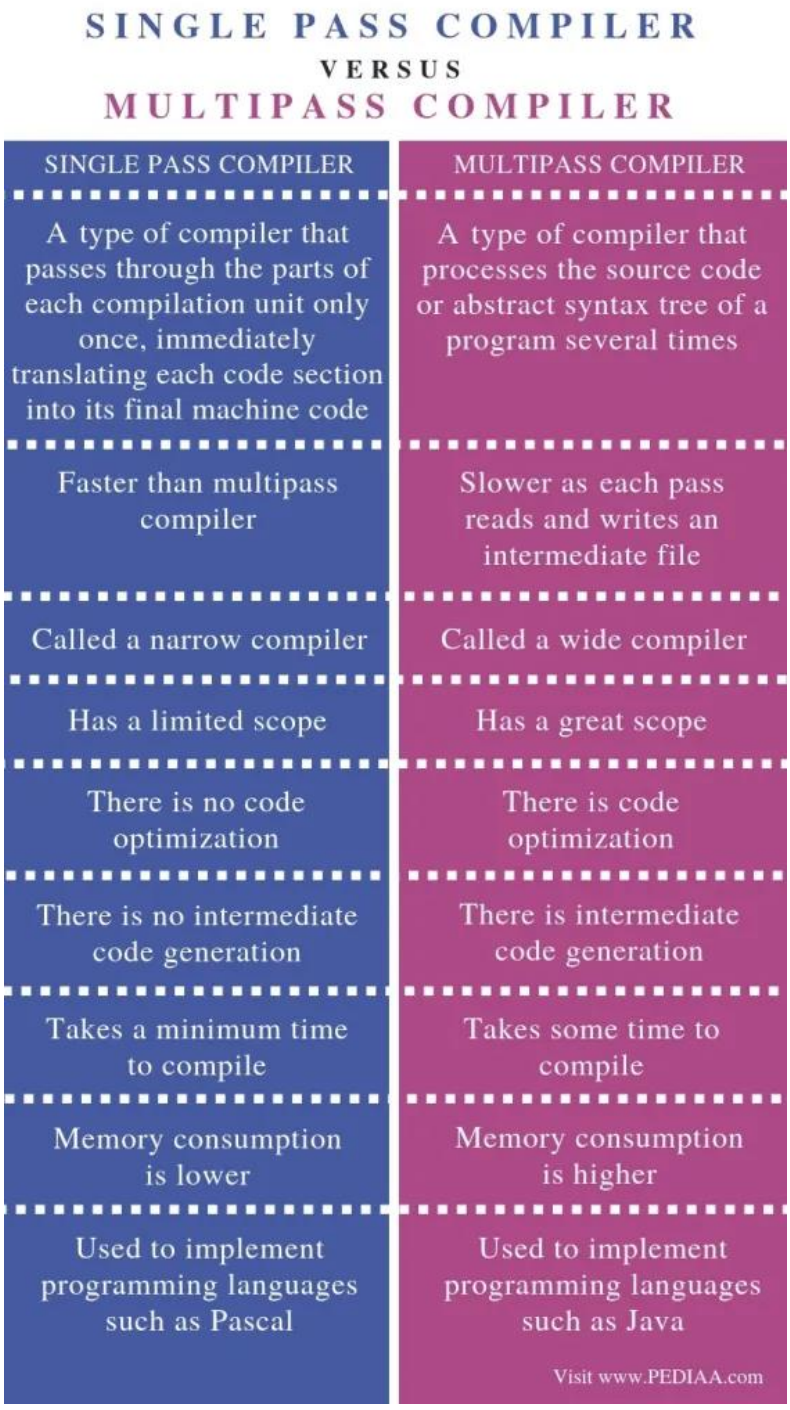


FIG 4.1: Compiler phases comparison - Source: https://pediaa.com/

4.2. Diagrams

- **Translators (Compilers).** T-shaped (T-shirt) tombstone represents translators (compilers). The head of the tombstone shows the translator's source language ***S*** and the target language ***T*** by

using one specific language ***L***. The arrow indicates the direction of the translation. The base of the tombstone shows the translator's implementation language.

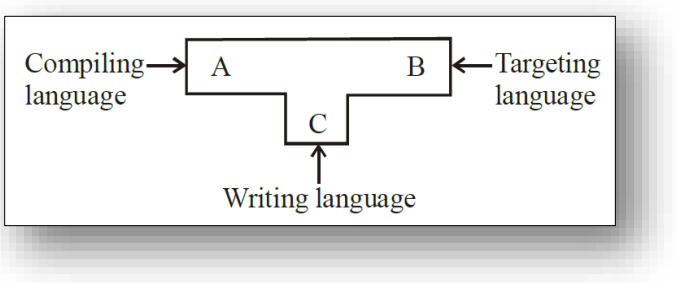


FIG 4.2: T Translator - Source: Aho (old version)

- For example: a Java-into-JVM code (Byte-code) compiler written in C, and a C++-into-C compiler written in C (C++ C-front compiler).

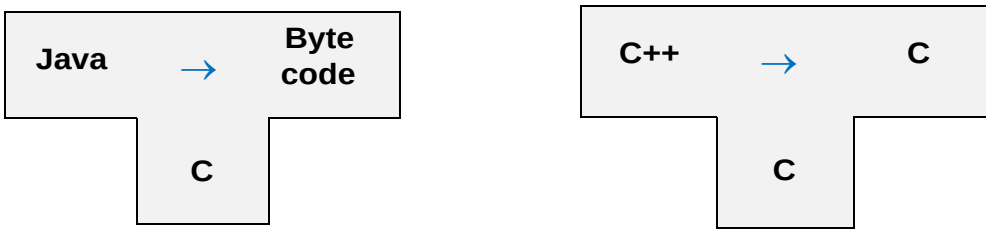


FIG 4.3: Example of T diagrams

- **Interpreters**. An interpreter is a program that accepts a program written in some source language, and in most cases, runs it immediately without translating the entire source program into target program (machine code). An interpreter is represented by a rectangular tombstone. The head of the stone indicates the interpreter's source language ***S***; the base shows the implementation language ***L***.
- For example: Java virtual machine (Java interpreter) expressed in x86, and BASIC interpreter written in C.

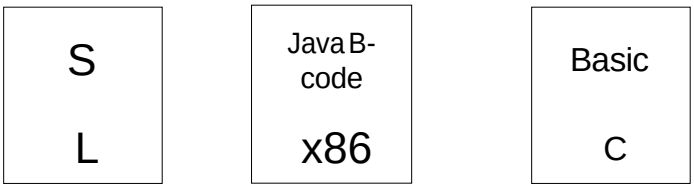


FIG 4.4: Example of Interpreters

4.3. Bootstrapping

- **Definition.** Is it possible to write a compiler using the same language the compiler is supposed to compile from? For example, to write a C compiler in C. The answer is "yes", and the process of writing such a compiler is called ***bootstrapping***. The idea is simple: First compiler is written

(in assembler or other language) that can compile a subset of the language.

- This compiler and the subset are used to write a compiler for the full language. This method is called ***full bootstrap*** because the whole compiler is to be written from scratch. The method can be applied ones or several times for richer and richer subsets of the language until a compiler for the full language is implemented.
- If a compiler for the language already exist but we want to write a compiler for a different machine using the same language the compiler compiles, the process is named ***half bootstrap*** since roughly half of the compiler must be modified.

4.4. Example of Hybrid compilation

Hybrid compilation is very well used. The general scheme is shown here:

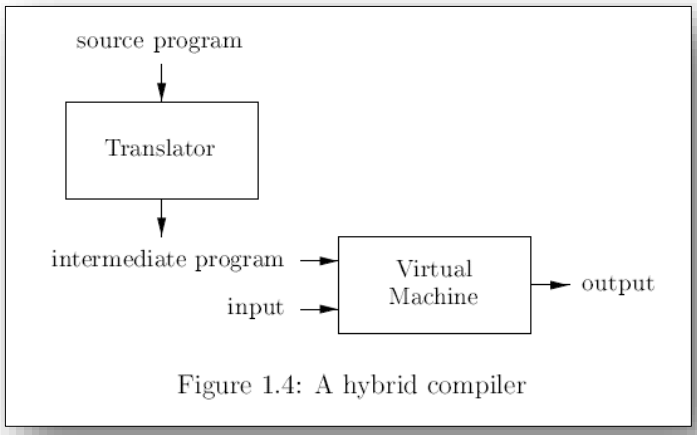


FIG 4.5: Hybrid compilation (Aho, main reference)

See this Java Code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

What is happening is that the Virtual Machine will operate and will generate this code (use the "javap -c" to decode the bytecode):

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1          // Method java/lang/Object."<init>":()V
       4: return
  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2          // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3          // String This will be printed
       5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Note that the compiler is responsible for executing all these operations.

4.5. And the Future?

- **To think about the future:**
  - Facebook AI Creates Its Own Language In Creepy Preview Of Our Potential Future:
    - https://www.forbes.com/sites/tonybradley/2017/07/31/facebook-ai-creates-its-own-language-in-creepy-preview-of-our-potential-future/
  - The truth behind Facebook AI inventing a new language:
    - https://towardsdatascience.com/the-truth-behind-facebook-ai-inventing-a-new-language-37c5d680e5a7
  - OpenAI API:
    - https://openai.com/blog/openai-api/

- **About automatic language for build compilers (GPT-3):**
  - GPT-3 Demo:
    - https://www.youtube.com/watch?v=8psgEDhT1MM
  - GPT-3 Paper:
    - https://arxiv.org/pdf/2005.14165.pdf
  - Kevin Lacker tests:
    - https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html

Recently (2021-2022):

- IBM has released the "**CodeNet**" service.
  - https://research.ibm.com/blog/codenet-ai-for-code

- **OPT: Open Pre-trained Transformer Language Models:**
  - https://wandb.ai/telidavies/ml-news/reports/Meta-AI-Releases-OPT-175B-Set-Of-Free-To-Use-Pretrained-Language-Models--VmlldzoxOTQwOTU1

- https://arxiv.org/abs/2205.01068

- **AI architecture Trends:**
  - Pathways: A next generation AI architecture (Oct 28, 2021): https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/

- **ChatGPT (OpenAI)[4]:**

OpenAI ChatGPT	➢ Basic Chat (Jan 6th 2023): https://openai.com/blog/chatgpt/

  - **ChatGPT** was fine-tuned on top of GPT-3.5 using **supervised learning** as well as **reinforcement learning**. Both approaches used human trainers to improve the model's performance. It uses the Proximal Policy Optimization (PPO) algorithms present a cost-effective benefit to trust region policy optimization algorithms and the models were trained in collaboration with Microsoft on their Azure supercomputing infrastructure.

Review Questions

1. How to define the compilers according to the passes to build programs.
2. Define the importance of bootstrapping.
3. Give real examples of bootstrapping.
4. What is the importance of hybrid compilation? What are their advantages / disadvantages?
5. What are the trends for compilers?

[4] See scientific papers that describe the ChatGPT – ex: *Training language models to follow instructions with human feedback* (https://arxiv.org/abs/2203.02155)

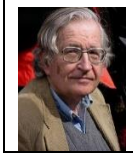
Article 5

Formal Languages

Languages are one important aspect of communication and, when using computers, it is necessary to define it formally (problems with ambiguity can happen)

5.1. CFG Grammars

5.1.1. HIERARCHY



Not all grammars are equal / equivalent. In fact, there is one hierarchy defined by Noam Chomsky (that was not a computer scientist, but a mathematical linguist, also know as the “father of formal languages”):

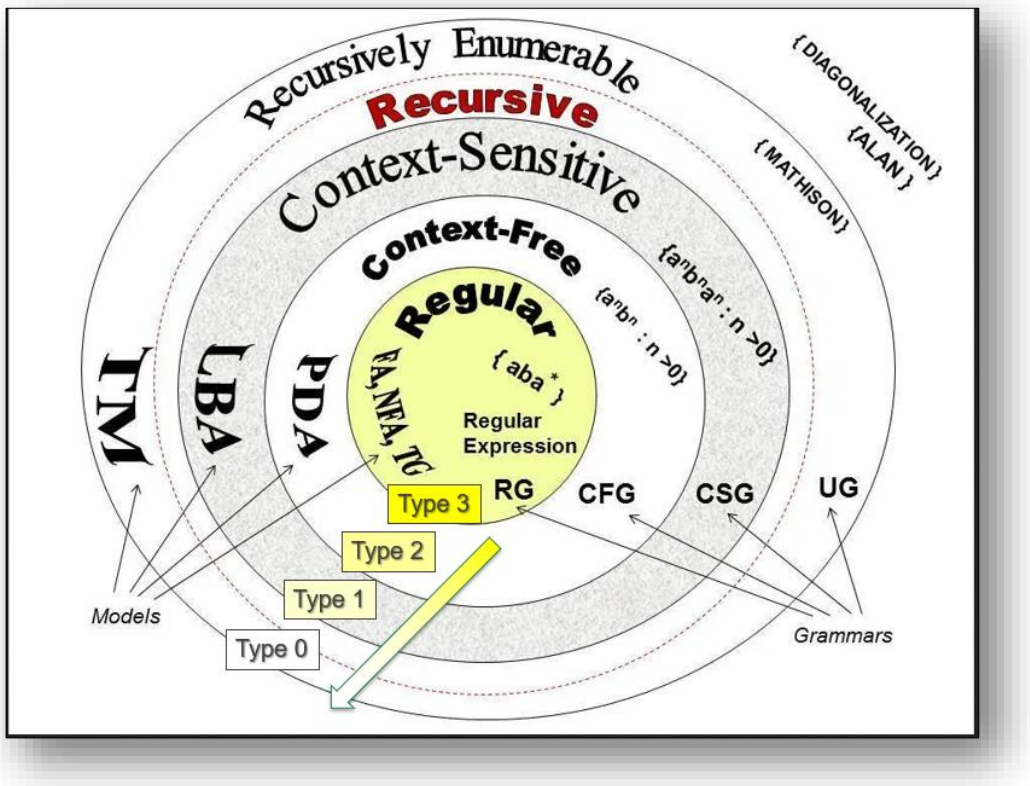


FIG 5.1: Chomsly language categories - Source: https://i2.wp.com/www.theoryofcomputation.co/wp-content/uploads/2018/09/Chomsky_Hierarchy.jpg

In short, we have:

Grammar	Languages	Model	Constrains	Example
Type-3	**Recursively enumerable**	Turing machine	$\gamma \rightarrow \alpha$	$L = \{w \mid w \in TM\}$
Type-2	**Context-sensitive**	Linear-bounded machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-1	**Context-free**	Push-down automata	$A \rightarrow \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-0	**Regular**	Finite state automata	$A \rightarrow a \mid aB$	$L = \{a^n \mid n \geq 0\}$

Inside, we have a generic model that can be “computable”:

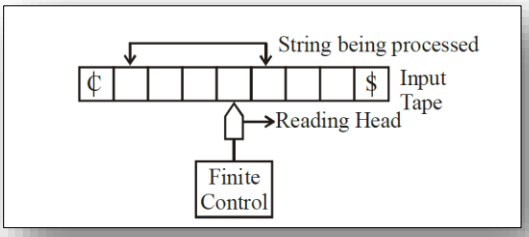


FIG 5.2: Diagram of a Finite Automaton

5.1.2. KLEENE’S THEOREM

In 1956 Stephen Cole Kleene formulated and proved the following theorem:

Theorem

The language that can be defined by any of these three methods

1. Regular Expressions

or

2. Finite Automaton (Finite State Machine)

or

3. Transition Graph.

But what is interesting is that other models are also equivalent. Ex: lambda calculus or formal grammars.

5.1.3. LANGUAGE BASICS

Noam Chomsky’s Grammar Hierarchy:

1. **Regular Grammars**
2. **Context Free Grammars (CFG or BNF)**
3. **Context Sensitive Grammars**
4. **General (Universal) Languages**

A **Context Free Grammar (CFG)** is defined by the following four components:

1. A finite set of terminal symbols (***terminals***) or a final terminal vocabulary **Vt**. For the lexical grammar the terminals are the alphabet; for the syntactic grammar the terminals are the token set produced by the scanner and defined by the lexical grammar.
2. A finite set of ***non-terminals*** or a nonterminal vocabulary **Vn** = Non-terminals are not part of the language. They are intermediate symbols used to define the grammar for the language.
3. A finite set of ***productions*** (rewriting or replacement or substitution or derivation rules) **P**.
4. Productions have the form:

   $A \rightarrow X_1X_2X_3\ldots X_m$

   Where:
   - $A \in V_n$, $X_i \in V_n \cup V_t$, $1 \leq i \leq m$,
   - $m > 0$ and
   - $A \rightarrow \varepsilon$ (empty) ($m = 0$) is a valid production

5. A **start** (or **goal**) symbol **S**. The start symbol $S \in V_n$ (S belong to V_n) is always the root of the parse tree.

Following the definition above, a CFG is the four-tuple **G = (Vt, Vn, P, S).**

- **Note**: L(G) is the **language** defined or generated by the grammar.

5.2. Regular Grammars

Grammar is a mathematical notation that operates on strings and uses the following operations:

- Concatenation (usually no sign), alternation (|), and recursion.

Examples:

- string1: a, string2: b
  - string1string2 ⇒ result: ab
  - string1 | string 2 ⇒ result: a or b ⇔ (a | b)
  - strings → string1 | stringsstring1
    ⇒ result: a or aa, or aaa or aaaa and so on…

- **Note**: Regular Grammars and be easily converted to Regular Expressions.

Grammar Examples (hypothetical language)

Here we are using one specific grammar for an hypothetical language (for instance, the old language created by prof. Svillen Ranev: PLATYPUS[5]):

<variable identifier> →
<arithmetic variable identifier> | <string variable identifier>
- Defined set: {a, b, c, a1, b12, c123, …, a$, b$, c$, a1$, b12$, c123$}

<arithmetic variable identifier> → <letter> <opt_letters or digits>
- Example: {a, b, c, …,abc, abcdf, abc123…}

<opt_letters or digits> → <letters or digits> | ε
- Example: {ε, a, b, c, …, abc, abcdf, abc123…}

<letters or digits> → <letter or digit> | <letters or digits> <letter or digit>
- Example: {a, b, c, … ,1, 2, 3, 1s, e2 …1111, aaaaa,…}

<letter or digit> → <letter> | <digit>
- Example: {a, b, c, … ,1, 2, 3 }

<letter> → a | b |…| z | A | B | …| Z
- Example: {a, b, c, … , z, A,…, Z }

<digit> → 0 | … | 9
- Example: {0, 1, 2, 3, …, 9 }

<string variable identifier> → $<arithmetic variable identifier>
- Example: {$a, $b, $c, …, $abc, $abcdf, $abc123…}

<integer literal> → <decimal integer literal>
- Example: {1, 2, , 0, 00, 111, 123, … , 777, …}

<decimal integer literal> → <zeros> | <non zero digit> <opt_digits>
- Example: {1, 2, , 0, 00, 111, 123, … , 777, …}

<zeros> → 0 | <zeros>0
- Example: {0, 00, 000, … , 00000000}

<opt_digits> → <digits> | ε

[5] The language we will use in labs is an adaptation of PLATYPUS, called "**Julius**" (in homenage to Julia - https://julialang.org/).

- Example: { ε , 1, 2, , 0, 00, 111, 123, … , 777, …}

<digits> → <digit> | <digits> <digit>

- Example: { 1, 2, , 0, 00, 111, 123, … , 777, …}

<digit> → 0 | <non zero digit>

- Example: { 0, 1, 2, … , 9}

<non zero digit> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Example: {1, 2, … , 9}

5.3. About Lambda

Initial Definitions

Alonzo Church idea: The lambda calculus (also written as λ-calculus, where lambda is the name of the Greek letter λ) was created by Alonzo Church in the early 1930s to study which functions are computable.

- In addition to being a concise yet powerful model in **computability theory**, the lambda calculus is also the simplest functional programming language.
- So much so that the lambda calculus looks like a "toy" language, even though it is (provably!) as powerful as any of the programming languages being used today, such as JavaScript, Java, C++, etc.

Fig. 5.3. Abstraction for functions (no internal state is important).

We just have:

- Variables
- Functions (how to define/apply)

We do not have:

- Datatypes
- Controls

Several definitions are functions:

- Constants
- Operations
- Expressions.

The Syntactical Grammar for Lambda Calculus

The syntax for lambda calculus is very simple:

```
<expression>   → constant
               | variable
               | (<expression> <expression>)
               | (variable.<expression>)
```

Where constants in this grammar are numbers or certain predefined functions like +, -, *, /. Variables are names like *x* or *y*.

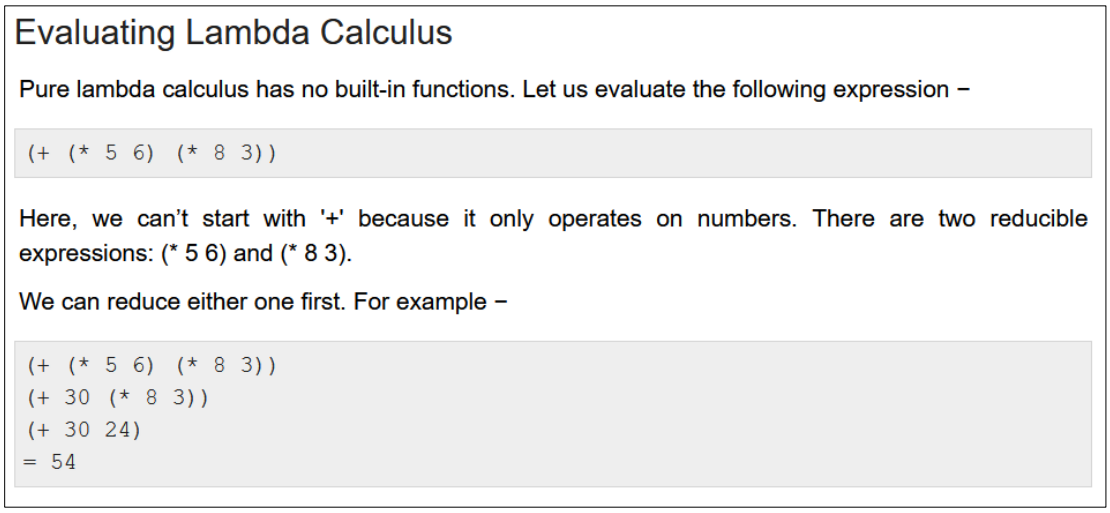
Example: (λx. (+1 x)) or (λx. + 1 x)

- This is a definition of a function that adds 1 to an arbitrary number x. It is called *lambda abstraction* and is represented by the 4^{th} production (or rule) in the grammar.
- The basic operation of the lambda calculus is the *application* of expressions such as the lambda abstractions.
- The expression (λx. + 1 x) 2 represents the application of the function that adds 1 to x to the constant 2. Lambda calculus provides a reduction rules that permits 2 to be substituted for x in the lambda abstraction and removing the lambda producing the value:

(λx. + 1 x) 2 => (+1 2) => 3

Additional Examples

Single Example:



Evaluating Lambda Calculus

Pure lambda calculus has no built-in functions. Let us evaluate the following expression −

```
(+ (* 5 6) (* 8 3))
```

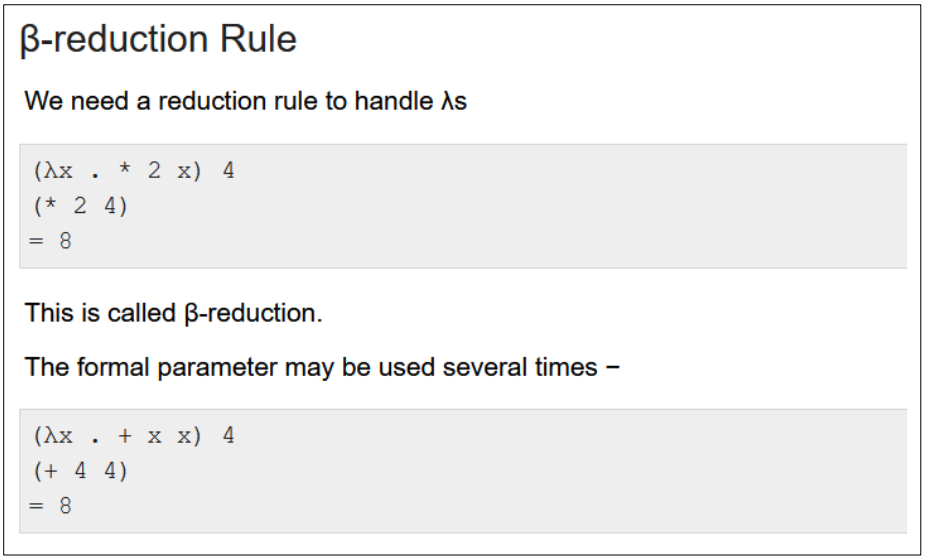
Here, we can't start with '+' because it only operates on numbers. There are two reducible expressions: (* 5 6) and (* 8 3).

We can reduce either one first. For example −

```
(+ (* 5 6) (* 8 3))
(+ 30 (* 8 3))
(+ 30 24)
= 54
```

FIG 5.3: Lambda examples (from Wikipedia)

Rules:



β-reduction Rule

We need a reduction rule to handle λs

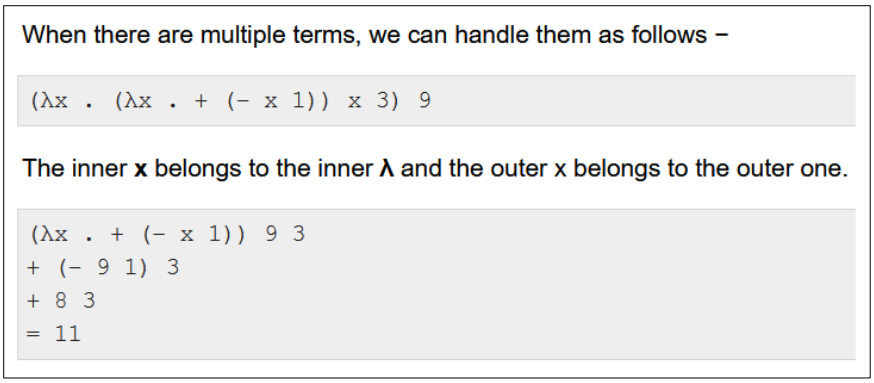
```
(λx . * 2 x) 4
(* 2 4)
= 8
```

This is called β-reduction.

The formal parameter may be used several times −

```
(λx . + x x) 4
(+ 4 4)
= 8
```

FIG 5.4: More examples (from Wikipedia)



When there are multiple terms, we can handle them as follows −

```
(λx . (λx . + (- x 1)) x 3) 9
```

The inner **x** belongs to the inner **λ** and the outer x belongs to the outer one.

```
(λx . + (- x 1)) 9 3
+ (- 9 1) 3
+ 8 3
= 11
```

FIG 5.5: Aditional examples - *Reference: Programming Languages, Principles and Practice, by Kenneth C. Louden.*

Example: Complex expressions (recursion – Haskell):

$$y = \lambda f.(\lambda x\ f(x\ x))\ (\lambda x.f(x\ x))$$

5.4. Using Boolean Functions in Lambda

Basic Definitions

Logical Interpretation:

- Basic values are used.
  - Ex: TRUE / FALSE
- Function Definitions:
  - **TRUE** = λx. λy . x
  - **FALSE** = λx. λy . y
  - **NOT** = λx.x FALSE TRUE
  - **AND** = λx.λy. x y FALSE
  - **OR** = λx.λy. x TRUE y
  - **XOR** = λx.λy. x (y FALSE TRUE) y
  - **IMPLIES** = λx.λy. x y TRUE
- Examples:
  - Example 1:
    - NOT TRUE =
      λx.x FALSE TRUE TRUE =
      (λx.λy.x) FALSE TRUE =
      FALSE
  - Example 2:
    - NOT FALSE=
      λx.x FALSE TRUE FALSE =
      (λx.λy.y) FALSE TRUE =
      TRUE

Review Questions

1. Why Kleene Theorem is important?
2. What is the advantage of RE representation?
3. What is the advantage of Grammar formalization?
4. Give other examples for RE and Grammars using your language.
5. What is the importance of lambda expressions?
6. What are the advantages of using lambda?

Article 6

Grammars and Languages

Computer languages have much in common with other human languages. Understanding their structure is the key to translating code written in them into an unambiguous set of instructions that can be accurately executed by a computing machine. The structure of a language is defined by its **GRAMMAR**.

6.1. Informal introduction

6.1.1. GRAMMARS

The informal rules for a syntactically correct English sentence as described in the book "The Language Instinct" are

Who Does What To Whom How Where When

or

Who Did What To Whom How Where When

This works in everyday life but is not very helpful if we want to develop a translator.

We need some well-defined notation allowing us to describe the syntactical rules of a language without any ambiguities. We need some type of metalanguage – a language for describing languages.

Consider the following sentence: *"The geeky student wrote the program"*.

In terms of the grammar of the English language this can be broken down into a **hierarchical parse tree:**

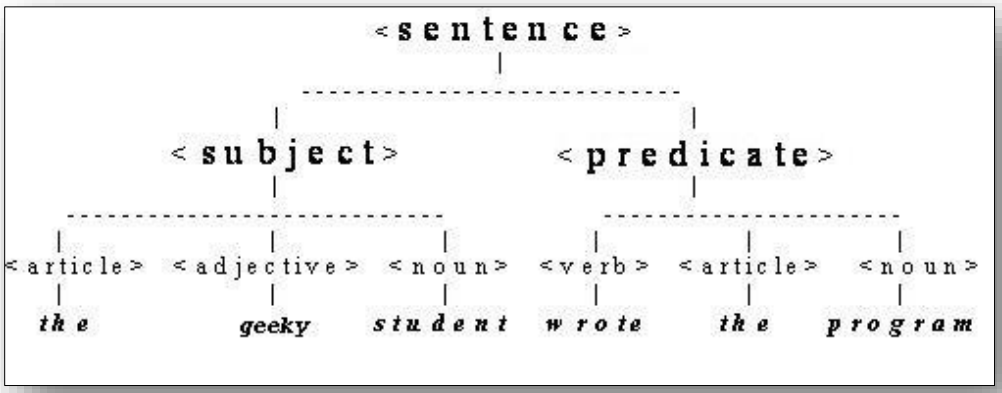


FIG 6.1: Examples of basic grammar

Note the use of meta-symbols like **<subject>** enclosed in <> to denote the syntactic entities. These constitute a special language that describes a language. Being a language about a language, it is a **metalanguage**.

The parse tree suggests a form for the metalanguage as a set of rules for the correct ways the syntactic entities may be formed. The following metalanguage is that of **Noam Chomsky** in a notation proposed by **John Backus**. Later we will use a more compact form called **Backus-Naur Form** (**BNF**).

The rules for the grammar appear to be

<sentence>	→	<subject> <predicate>
<subject>	→	<article> <adjective> <noun>
<predicate>	→	<verb> <direct object>
<direct object>	→	<article> <noun>
<article>	→	**the**
<adjective>	→	**geeky**
<verb>	→	**wrote**
<noun>	→	**student**
<noun>	→	**program**

Some notes about this formalism:

- The **metasymbol** ::= means “has the form of” or “may be composed of”.
- A single rule is called a **production** since what is on the left-hand side can produce a more detailed string on the right-hand side.
- The **metasymbols enclosed** in <> (<sentence> etc.) are called **non-terminals** since they will be replaced by applying the production.
- The symbols in **bold** (the, geeky, etc.) are called **terminals** since they terminate the syntax (they are the leaf nodes).
- The set of terminal symbols forms the sentence that we finally recognize:

"The geeky student wrote the program"

BNF allows an abbreviation of alternative productions:

<sentence>	<subject> <predicate>
<subject>	<article> <adjective> <noun>
<predicate>	<verb> <direct object>
<direct object>	<article> <noun>
<article>	**the**
<adjective>	**geeky**
<verb>	**wrote**
<noun>	**student | program**

6.1.2. LANGUAGES

A language is the set of sentences that are generated from the grammar. So given the grammar above, possible sentences are:

1. "The geeky student wrote the program"
2. "The geeky program wrote the student"

The second sentence does not look right (unless "student" is the name of output of the program) and therefore fails to be semantically correct, even though it is syntactically correct according to the grammar.

6.2. Scanning and Parsing (Lexical and Syntactic Analysis)

6.2.1. INTRODUCTION

Given a string of characters, how do we know it is a valid sentence according to the grammar? There are two issues here:

1. **Lexical analysis (the scanner)** - constructing the string of tokens (terminal symbols) from the string of characters
2. **Syntactic analysis (the parser)** – validating (recognizing) the token string against the grammar.

This division is not really so precise since parsing in its general meaning is also part of the scanning process.
In **lexical analysis** (the **scanner**) we attempt to collect the incoming characters into tokens, or terminal symbols of the grammar. So, the following string of characters:

"The geeky student wrote the program"

generates the tokens ARTICLE, ADJECTIVE, VERB, NOUN (represented in the program code as symbolic constants, possibly with the values 0, 1, 2, 3), with their lexical values **the**, **geeky**,

wrote, **student**, **program**. So, when the input stream has been scanned we end with the symbol table containing the entries:

token	attribute - lexeme
ARTICLE	the
ADJECTIVE	geeky
NOUN	student
VERB	wrote
ARTICLE	the
NOUN	program

- **NOTE**: In a real programming language, the attribute may be a string of arbitrary length called a lexeme, pointed to by the attribute pointer).

However, the task of the scanner is also to reject an input string like: "*Thr gyeke stuent wnote te pgram*", since no tokens can be identified.

- Therefore, the task of the scanner in lexical analysis is to parse the input stream of characters, group them (into lexemes) and recognize them as tokens matching the patterns associated with the tokens.
- In doing this we will use a simpler grammar for describing the lexemes called a regular grammar for which is equivalent to regular expressions notation for describing strings (lexemes).

In **syntactic analysis**, the role of the **parser** is to confirm (recognize) that the sentence of tokens satisfies the grammar. This is parsing the token stream.

Suppose the scanner has recognized the input: "The geeky student wrote the program" as the string of tokens:

ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN

- The syntactic analyzer (parser) must then confirm that it is valid grammatically by using the productions of the grammar to construct (usually implicitly) a syntax tree. If the tree fails, then the string of tokens is not a sentence of the language.
- This can be done by top-down **or bottom-up parsing** technique.

6.2.2. TOP-DOWN PARSING

In this we begin with the <sentence> nonterminal, called the start symbol, and by successive applications (derivations) of the productions attempt to arrive at the sentence. At each stage one token is consumed. Here are the productions, labeled for identification:

Non-terminal	Production Rule
<sentence>	<subject> <predicate>
<subject>	<article> <adjective> <noun>

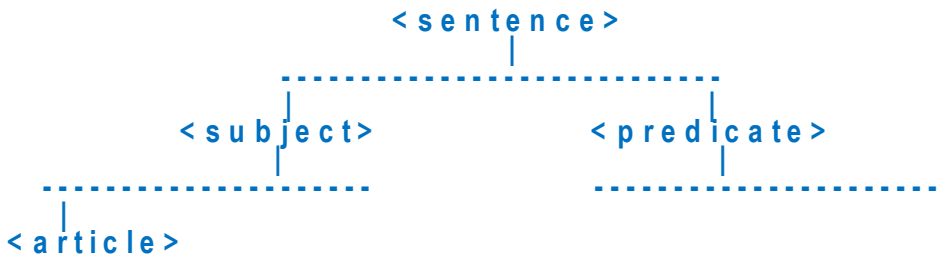
<predicate>	<verb> <direct object>
<direct object>	<article> <noun>
<article>	**the**
<adjective>	**geeky**
<verb>	**wrote**
<noun>	**student | program**

Starting from the left of the token string (a **left-most derivation**) we look at the first token **ARTICLE**.

- **NOTE:** The ? indicates our position in the token stream (the token currently being inspected is called the **lookahead**):

ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN
?

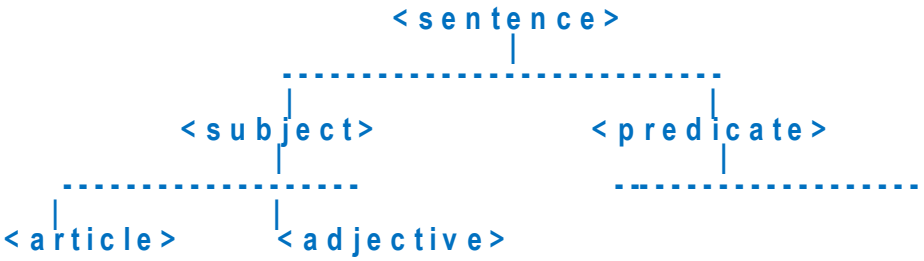
We begin at the start nonterminal <sentence>. We find using rule 1. that a sentence starts with a <subject> that starts with an <article>. So, the parse tree at this point is:



Now that **ARTICLE** has been recognized, it is consumed, the token string is shorter, and we are now attempting to recognize **ADJECTIVE**

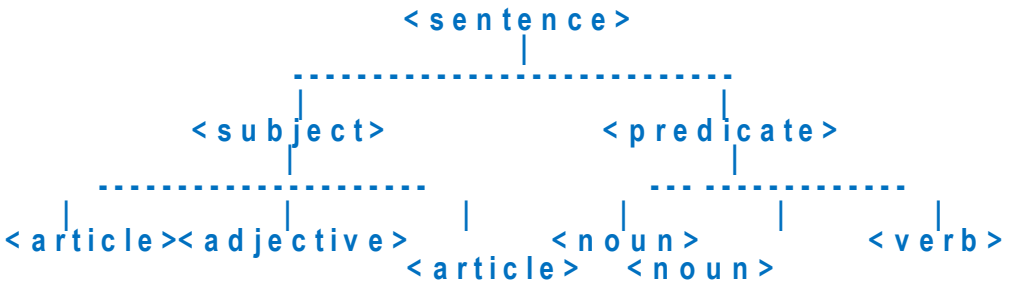
ADJECTIVE NOUN VERB ARTICLE NOUN

We see from rule 2 that a <subject> is an <article> followed by an <adjective>. So **ADJECTIVE** is consumed, and the parse tree is:



and <adjective> is consumed.

We continue in this way until all the input tokens are consumed and the parse tree builds the token string in order:



The input token string has been successfully parsed and is grammatically valid.

Note that because we used the scanner to do a prior lexical analysis, the actual lexemes do not appear in this tree. If we had included the scanner in the complete process, the complete tree would have been:

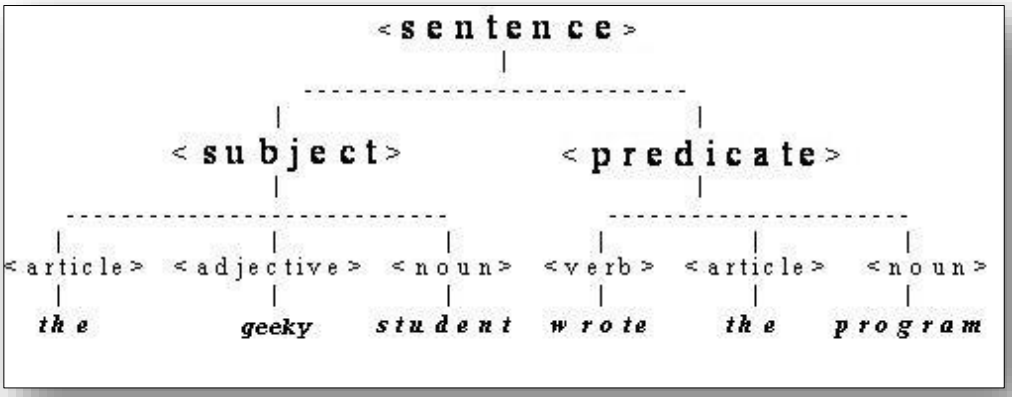


FIG 6.2: Reviewing examples

Once we have defined the grammar for a sentence, we can define the grammar for a novel (program)

<novel>	→	**<sentences>**
<sentences>	→	**<sentence> <sentences> | <sentence>**

Since the nonterminal <sentences> appears on both side of the production it means that a <sentence> is **self-describing**. The <sentences> is described in terms of itself and is therefore **recursive.**

Humans express themselves recursively in written, spoken and programming languages. Useful programming languages are also recursive and can use recursive algorithms to implement the parser (**recursive descent parser**). However, recursion will cause problems in the code implementation of the grammar that we will need to solve.

6.3. Example

Informal Specification of the Calc Language

The Calc language is a language for programming simple calculations with whole numbers.

- The Calc program is a sequence of ***assignment statements***.
- An assignment statement consists of an identifier representing a variable followed by an **=** symbol, followed by an ***arithmetic expression.***
- An ***arithmetic expression*** is an infix expression constructed from ***variables***, ***integer literals***, and the ***operators*** *plus* (**+**), *minus* (**-**), *multiplication* (*****), and *division* (***/***).
- The arithmetic expression always evaluates to an integer value which is assigned to the variable on the left side of the = sign.
- If a variable is used in an expression, the variable must have a previously assigned value.
- Before termination, the program automatically prints the values of all variables introduced in the program.

Example of a Calc program:

```
a = 1 * 5
b = a / 3
c = a + b * c / d
```

Output:

```
a → 5
b → 1
c → 6
```

Lexical Grammar for the **Calc** Language

```
<vid>       →   <letters>
<letters>   →   <letter> | <letters><letter>
<letter >   →   a | b | … | z
<num>       →   <digits>
<digits>    →   <digit> | <digits> <digit>
<digit>     →   0 | 1 | 2 | … | 9
<operators> →   + | - | * | / | =
```

Syntactical Grammar for the Calc Language (Calc Syntax)

```
<program>      →    <statements>
<statements>   →    <statement> <statements> |<statement>
<statement>    →    <assignment>
<assignment>   →    vid = <expression>
<expression>   →    vid
                  | num
                  | <expression> + <expression>
                  | <expression> * <expression>
                  | <expression> - <expression>
                  | <expression> / <expression>
```

6.4. About Parse Trees

Examples:

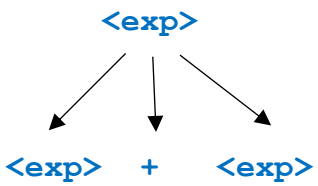
- Grammar 1: Ambiguous

```
<exp> → vid
    | num
    | <exp> + <exp>
    | <exp> * <exp>
    | <exp> - <exp>
    | <exp> / <exp>
```

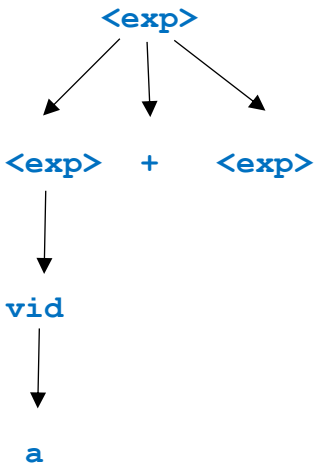
Example: a + 2 * b

TREE 1

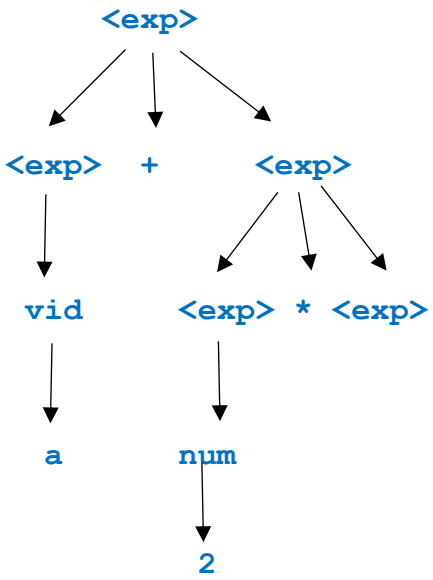
Step 1:



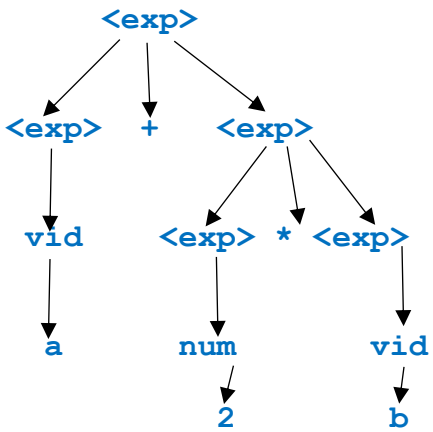
Step 2:


<exp>
<exp> + <exp>
vid
a

Step 3:

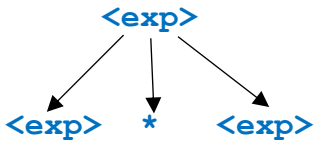

<exp>
<exp> + <exp>
vid
<exp> * <exp>
a
num
2

Step 4:

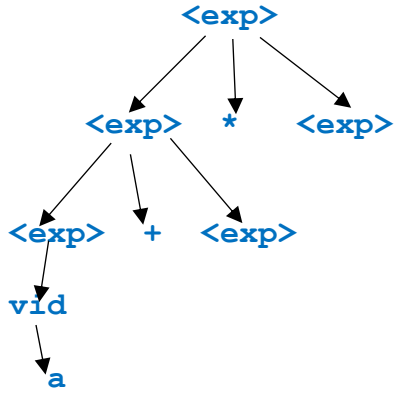

<exp>
<exp> + <exp>
vid
<exp> * <exp>
a
num
vid
2
b

TREE 2

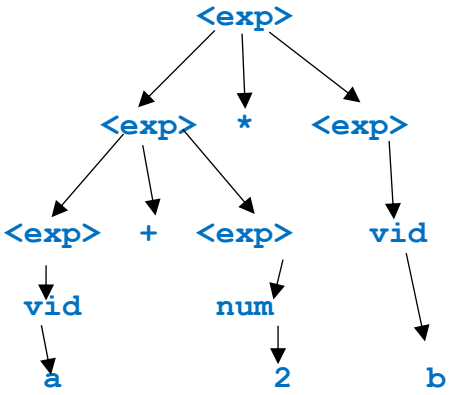
Step 1:



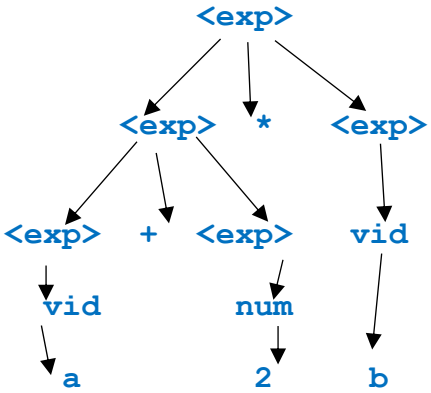
Step 2:



Step 3:



Step 4:

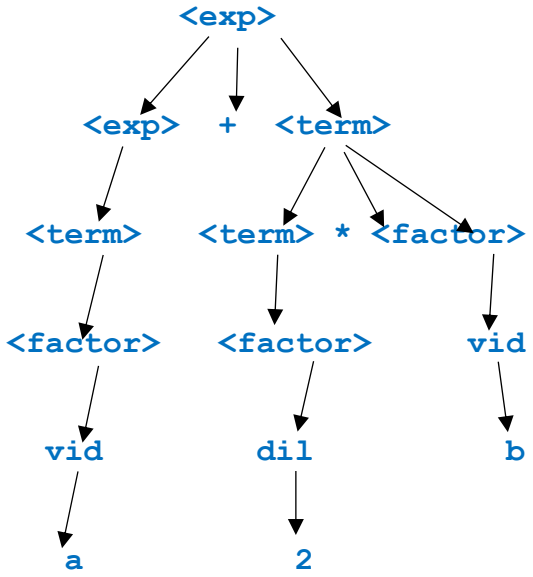


- Grammar 2: No-Ambiguous

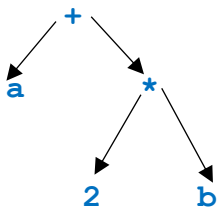
```
<exp>  →  <exp> + <term>
       | <exp> - <term>
       | <term>

<term> →  <term> * <factor>
        | <term> / <factor>
        | <factor>

<factor> →      vid | dil | fpl | (<expression>)
```

Example: `a + 2 * b`

Parse tree



Syntax tree

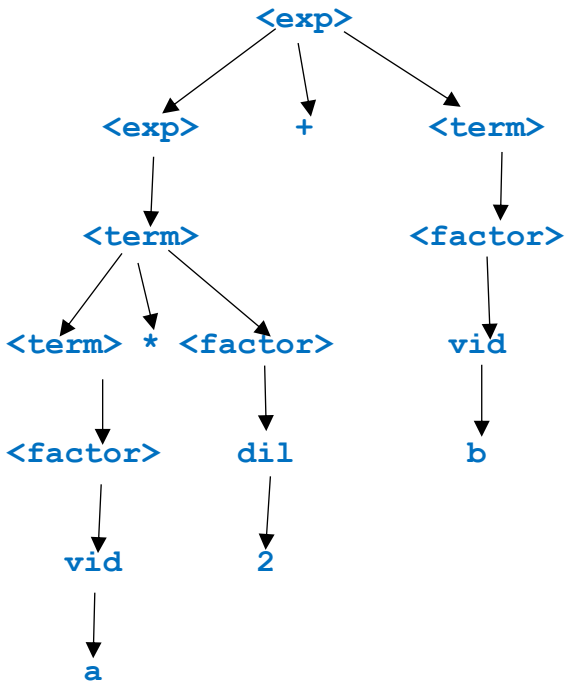


- Grammar 3: More detailed grammar

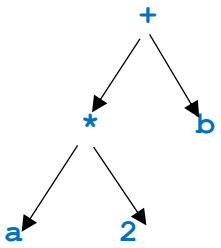
```
<exp> → <exp> + <term>
      | <exp> - <term>
      | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → vid | dil | fpl | (<expression>)
```

Example: `a * 2 + b`

Parse tree



Syntax tree



6.5. Reviews

Ambiguity

A grammar is said to be **ambiguous** if there exists more than one left-most derivation or more than one right-most derivative or more than one parse tree for the given input string.

- If the grammar is not ambiguous then it is called **unambiguous**.
- Note: we all expect that to create formal languages, we just have unambiguous grammar.

Types of Parsers

- **Top down parsing**
  - The parsing starts from the start symbol and transform it into the input symbol.
  - Also known as **recursive parsing** or **predictive parsing**.
- **Button up parsing**
  - The parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the **right-most derivations** of string in **reverse**
  - It is used to construct a **parse tree** for an input string.
  - Also known as **shift-reduce** parsing.

Review Questions

1. Using your words explain the importance of grammars.
2. How be a grammar can defined?
3. What is the problem with ambiguous grammar?
4. How to solve grammar ambiguity?
5. What is the importance of lambda calculus?

.

Article 7 Regular Expressions

7.1. Regular Expressions Basics

Regular expressions are a convenient notation (or means or tools) for specifying certain simple (though possibly infinite) set of **strings** over some **alphabet**.

- As such, they can be used for describing lexical symbols or for specifying the structure (pattern) of the token used in a programming language.
- **TIP**: A regular expression can be used to construct a Deterministic Finite Automaton (DFA) which therefore can recognize strings (words) of the grammar, which is the purpose of the Scanner.

A regular expression is a shorthand **equivalent** to a regular grammar; it is a pattern that strings must match or conform to, to be valid words (or sentences).

- The sets of strings defined by regular expression are termed **regular sets**.

As defined above, the grammar is the following tuple: G = (Vt, Vn, P, S). As shown in the Grammar formal specification, any grammar must contain terminals, non-terminals, production rules and start symbol. If we denote r as a regular expression, then since G and r are equivalent, they must produce the same languages, so:

L(G) = L(r)

To define the strings, regular expressions (as any expression notation) use operands and operations.

- The operands are alphabet symbols or strings defined by regular expressions (regular definitions).
- The standard operations are catenation (concatenation), union or alternation (|), and Kleene closure (*)
- Regular expressions use the **metasymbols** |, (,), {, } , [,], * , + (and others ?, ^) to define its operations.

7.1.1. Alphabet

An **alphabet** ∑ is a finite, non-empty, set of symbols. For example:

- the binary alphabet is {0, 1}

- the decimal alphabet is {0,1,2,3,4,5,6,7,8,9}
- **Note**: The metasymbols { , and } used here that are not in the alphabet.

IMPLEMENTATION NOTES:

- For the scanner, the alphabet may be characters in the ASCII character set.
- For the parser the alphabet is the set of tokens produced by the scanner.
- The set of keywords is also an alphabet (even though by itself it is insufficient to form a language):
  - **Hypothetical** keywords: { **data, code, int, float, string, if, then, else, while, do** }
    - **Note:** Functions (ex: main&, read&, write&) are not considered "keywords".

7.1.2. String

A string is a finite set of symbols from an alphabet (not necessarily in a grammar).

For example: For the alphabet ∑ = {a, b, c} some possible strings are:

{a, b, c, ab, ac, bc, ba, ca, cb, abc, acb, bac, bca, cab, cba}

- **Note:** The order of symbols in a string matter.

Empty string

ε is the **empty string**. It is the string consisting of no symbols.

The **length of a string** s is |s| and is equal to the number of symbols in the string.

So: |ε| = 0, |abc| = 3.

Single Character Regular Expressions

a. A single character.

A regular expression can be a pattern for a single character from the alphabet . This is the simplest case. Consider an alphabet

∑ = {a, b, c, d}.

If we write the regular expression b, then:

L(b) = {b}

- In plain language, the regular expression b means simply the set of characters consisting only of b, which means that the regular expression b is the character b.
- On the same token, a string s is regular expression denoting a set containing only s. if it contains meta-characters, s can be quoted to avoid ambiguity ("s").

b. The empty string

A regular expression can be the empty string ε. This is defined by:

$$L(\varepsilon) = \{\varepsilon\}$$

The regular expression **ε** means the set containing only the empty string.

c. The regular expression that matches nothing: Φ.

Defined by Φ, it is the pattern for nothing; it generates the set containing nothing

$$L(\Phi) = \{\ \}$$

- **Note:** This is not the same as the empty string.

By contrast, ε is the pattern for the set that contains the string that contains no characters.

Concatenation of strings

Concatenation joins the strings together.

- For example: If x and y are strings, then their concatenation is the string xy.
- For the alphabet {a, b, c}, if x = ab and y = bc, then xy = abbc
- Concatenation with the empty string ε leaves a string unchanged:

$$\varepsilon x = x\varepsilon = x$$

7.1.3. Operations

Operations with Regular Expressions

a. Alternation – the | operator

If x and y are regular expressions, then x | y is the regular expression with alternatives. It means a string that matches either the regular expression x or the regular expression y. Formally:

$$L(x \mid y) = L(x)\, L(y)$$

For example if ∑ = {a, b, c, d} then:

L(a|b) = {a,b} and L(c|d) = {c,d} so a|b is either a or b

c | d is either c or d

Likewise, the regular expression a | ε s either a or ε.

b. Concatenation (writing adjacent letters or strings, or . or ,)

The concatenation of two regular expressions x and y is xy. It means take one of x's string followed by one of y's strings. Formally:

$$L(xy) = L(x)L(y)$$

More complex expressions can be written.

Example 1:

(a|b)c

The parenthesis enforces alternation before concatenation.

L((a|b)c) = L(a|b)L(c) = {a,b} {c} = {ac, bc}

Example 2:

(a|b)(a|b)

The parenthesis enforces alternation before concatenation.

L((a|b)(a|b) = L(a|b)L(a|b) = {a,b} {a,b} = {aa, ba, ab, bb}

Example 3:

aa|ba|ab|bb

This is just another way of writing example 2

c. Recursion - Kleene Closure

For a regular expression r, Kleene Closure is defined by:

$$L(r^*) = L(r)^*$$

which means concatenation with all powers of L. As an example, from our alphabet,

$$L((a|bb)^*) = L(a|bb)^* = L(a|bb)^0 + L(a|bb)^1 + L(a|bb)^2 + L(a|bb)^3 + \ldots$$

And

$L(a|bb)^0 = \{\varepsilon\}$
$L(a|bb)^1 = \{a,bb\}$
$L(a|bb)^2 = \{a,bb\}\{a,bb\} = \{aa, abb, bba, bbbb\}$
$L(a|bb)^3 = \{aa, abb, bba, bbbb\}\{a,bb\} =$
$\{aaa,abba,bbaa,bbbba,aabb,abbbb,bbabb,bbbbbb\}$
so
$L((a|bb)^*) = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, abba, bbaa, bbbba, aabb, abbbb,$
$bbabb, bbbbbb,\ldots\}$

Nonstandard Regular Expressions operations

Positive Closure (One or more +)

$a^+ = aa^*$ (also $a^* = a^+| \varepsilon$)

Exponentiation or Power operation (exactly k)

$a^k = aaa\ldots a$ (exactly k times) Optional

Inclusion (Zero or one ?)

$a? = \varepsilon \mid a$

Character classes

- Specify a **range** of characters or numbers that follow a sequence.
- Examples:
  - [a-z] means any character in the range a to z.
  - [A-Z] means any character in the range A to Z.
- A regular expression for the **pattern** for an identifier that begins with a letter or an underscore and is followed by any number of numbers and letters is

[a-zA-Z_][A-Za-z0-9]*.

A **complement character class** is specified using the ^ or (~) symbols, or **Not** operator [^a-z] matches any character except a to z.

Comment = // (^CR[6])*CR

[6] CR = Carriage Return

Precedence of Operators

The parentheses, ( ) allow us to override the precedence of operators. In order of decreasing precedence:

- Grouping: ( )
- Character classes: [ ]
- Power / Recursion (Kleene star): *
- Concatenation: .
- Alternation: |
- Positive closure: +
- Optional: ?
- Iteration: k

Operations on Sets of Strings

a. Concatenation of sets

The concatenation of two sets A and B is defined by:

AB = { xy | x in A and y in B }

which reads "*the set of strings xy such that x is in A and y is in B*".

For example.

If A = {a,b} and B = {c,d} then AB = { ac, ad, bc, bd }

b. Powers of sets

The power of a set A: The repetition of A several times.

A^4 = { x | 4-symbol string }

which reads "*the set of strings with four symbols*".

This is just repeated concatenation:

A^0 = { ε }, A1 = A, A2 = AA, A3 = AAA, …

- Note that A^0 = { ε } for any set)

For example.

If A = {a,b} then A^0 = { ε }, A^1 = { a, b }, A^2 = { aa, ab, ba, bb }, A^3 = { aaa, aab,aba, abb, baa, bab,

bba, bbb }…

c. Union of sets

The union of two sets A and B is defined by:

$$A \cup B = \{ x \mid x \text{ in } A \text{ or } x \text{ in } B \}$$

which reads "*the set of strings x such that x is in A or x is in B*".

For example.

If A = {a,b} and B = {c,d} then A ∪ B = { a, b, c, d }

d. Kleene Closure

The Kleene closure of a set A is the * operator defined as the set of all strings including the empty string:

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

It is the union of all powers of A.

$$A^* = A^0 + A^1 + A^2 + A^3 + \ldots$$

For example:

If A = {a,b} then:

A* = { ε } + { a,b } + { ab,ba,aa,bb } + { aaa, aab,aba, abb, baa, bab, bba, bbb } +…
= { ε, a, b, ab, ba, aa, bb, aaa, aab,aba, abb, baa, bab, bba, bbb, …}

e. Positive Closure

The positive closure is the + operator defined as the set of all strings excluding the empty string:

$$A^+ = \bigcup_{i=1}^{\infty} A^i$$

It is the union of all powers of A except the empty string.

$$A+ = A* - \{\varepsilon\} = A^0 + A^1 + A^2 + A^3 + \ldots$$

For example:

If A = {a,b} then: A+ = { a,b } + { ab,ba,aa,bb } + { aaa, aab,aba, abb, baa, bab, bba, bbb } +… = { a, b, ab, ba, aa, bb, aaa, aab,aba, abb, baa, bab, bba, bbb, …}

Kleene closure is the union of the set containing the empty string and positive closure:

$$A* = A+ \cup A^0$$

Examples

Let L be the set {A, B, … Z, a, b, … z} and let D be the set {0, 1, …….9}

- L ∪ D is the set of letters and digits: {A, B, … Z, a, b, … z, 0, 1, …….9}
- LD is the set of strings consisting of a letter followed by a digit: {A0, A1, A2 .., B0, B1,… Z9, a0, b0, a1, b1 … }
- L^4 is the set of all four-letter strings
- L* is the set of all strings of letters, including ε.
- L(L∪D)* is the set of all strings of letters and digits beginning with a letter
- D+ is the set of all strings of one or more digits

7.1.4. Formal Definition of Regular Expressions

General Definitions

Each regular expression denotes (defines) a set of strings (regular sets). Formally, regular expressions are defined as follows.

- Φ is a regular expression denoting the **empty set**.
- ε is a regular expression denoting the set that contains only the **empty string**.
- A **string s** is a regular expression denoting a set containing only s.
- If A and B are **regular expressions**, then A | B, AB, and A* are also regular expressions. In this case, A and B are sometimes called regular definitions

Regular expressions define sets of strings. Regular expression operations are operations on sets of strings.

Limitations of regular expressions

Regular expressions are simpler than grammars but are less powerful. Some patterns can only be derived from grammars (matched parenthesis, repeating patterns).

- **Note**: Every pattern that can be described by a regular expression can also be described by a grammar.

Example

Regular Expression:

(a|b)*abb

- a or b, repeated zero or more times, followed by abb:

{abb, aabb, aaabb, …, babb, bbabb, bbbabb…}

Equivalent grammar:

A0 → aA0 | bA0 | aA1
A1 → bA2
A2 → bA3
A3 → ε

Regular expressions laws

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

FIG 7.1: Examples of basic grammar

RE Examples

VID = AVID | SVID

- Defined set {a, b, c, a1, b12, c123, …, a$, b$, c$, a1$, b12$, c123$}

AVID = L (L | D)*

- Examples: {a, b, c, ….abc, abcdf, abc123…}

L = a | b |…| z | A | B | …| Z

- Examples: {a, b, c,…z, A,…, Z }

D = 0 | … | 9

- Examples: {0, 1, 2, 3,…9 }

SVID = $AVID

- Examples: {$a, $b, $c, ….$abc, $abcdf, $abc123…}

IL = DD*

- Examples: {1, 2, , 0, 00, 111, 123, … , 777, …}

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}\,s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
r_1r_2	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: Lex regular expressions

FIG 7.2: RE examples

Review Questions

1. What is the advantage of RE representation?
2. What is the advantage of Grammar formalization?
3. Give other examples for RE and Grammars in your language.

Article 8 Finite Automata

8.1. Definitions

8.1.1. FINITE AUTOMATON

A **Finite Automaton (FA)** or **Finite State Machine (FSM)** is consists of the following components (parts):

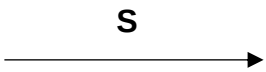
- A **finite set of states**, one of which is designated as the initial state, called the **start state**, and some states of which are designated as final states, called also halting states, terminal states or accepting states.
- An **alphabet** Σ of possible input symbols (including ε), from which are formed the strings (words) of the language recognized by the FA.
    - A **finite set of transition** that determine for each state and for each symbol of the input alphabet which state to go next. In other words, the FA must have a transition function **move** or **next_state** that maps each state-symbol pair to a single state or set of states.
    - If the function maps each state-symbol pair to a set containing only one state, the FA is defined as Deterministic Finite Automaton (DFA).
    - If the function maps each state-symbol pair to a set containing more than one state, the FA is defined as **Nondeterministic Finite Automaton (NFA)**.
    - **Note**: NFA can be easily constructed from regular expressions and then converted to DFA. Each NFA can be converted to DFA.
- The set of transitions can be represented in different ways but the most common and the easiest for implementation in computers is the transition table representation.
    - The transition table has one row for each state and one column for each input symbol.
    - The entry for row i and column s is the state (for DFA) that can be reached by transition from state i on input symbol s.
    - The entry for row i and column s is the states (more than one state for NFA) that can be reached by transition from state i on input symbol s.
    - The transition table representation has the advantage that it can be build directly from the corresponding transition diagram and it provides very fast access; its disadvantage is that it takes a lot of space.
    - A compromising solution is to compress the table to save space and make the access a bit slower.

Since the FA accept or recognize strings of a language, they are also called finite acceptors or language recognizers.

- A **recognizer** is able to take a string 'x' and determine whether the string belong to the language defined by the FA.
- A **regular expression** can be turned into a recognizer by making a transition diagram or transition table which represents a finite automata
- An **automaton** can be considered either deterministic or non- deterministic
- Both types of automata can be used in developing a lexical analyzer
- Algorithms involving automata exist which can help improve the design of scanners

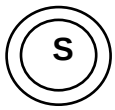
8.1.2. EXAMPLES

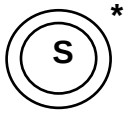
- **Transition or state diagrams**



- **State**

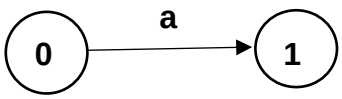
- **Accepting states** (called also halting states or terminal states)



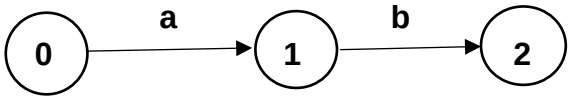


(Using retract operation)

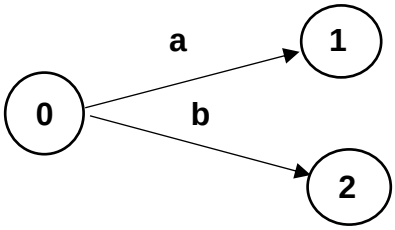
- **Regular expression: a**



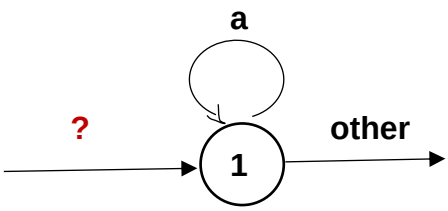
- **Regular expression: ab (concatenation)**



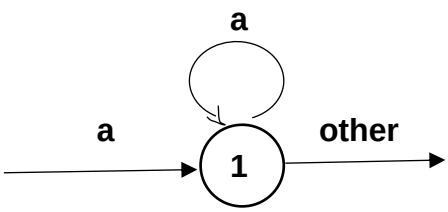
- **Regular expression: a | b (alternation)**



- **Regular expression: a* (Kleene Closure)**



- **Regular expression: a+ (Positive Closure)**



8.1.3. NFA – NON-DETERMINISM

- A DFA is a special case of an NFA, except for the following restrictions:
  - No empty ε-transitions exist within it
  - Every transition from a state is unique
- An NFA can always be converted into a DFA; however, the number of states will increase.
- A DFA will often be faster than an NFA, because decisions regarding transitions don't have to be made.

8.2. Implementing Automaton

- A transition diagram can be implemented manually, with states being represented by variables, and transitions being decided on by if statements.

- Example:
  - Consider a procedure which recognizes **floating point** and **integer** constants.
  - Transitions are indicated by arrows with a input symbol label, states are indicated by circles with unique labels and final states are indicated by two concentric circles with unique labels. * indicates that the input must be retracted – the last symbol read must be returned back to the input stream of symbols.

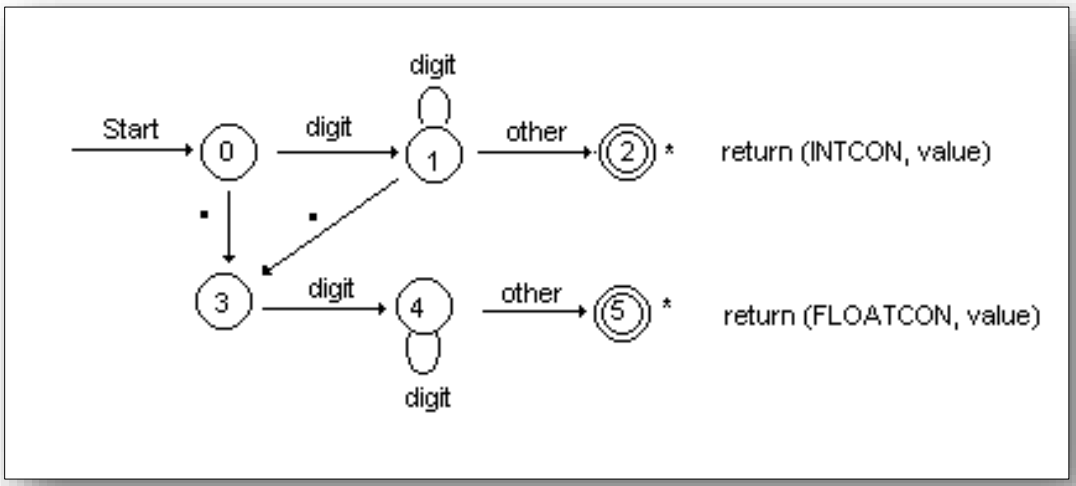


FIG 8.1: Initial automaton to numbers

  - Cases would be developed which handle all states, and all transitions from one state to another.

8.2.1. Example of Transition Diagram Implementation:

- The following (partial) code implements the previous transition diagram:

```
token next_token() {
      while (1) {
            switch(state)
                  case 0:
                        c=nextchar();
                        if (c == ' ') {        // Skip blanks
                              lexeme_start++;
                              state = 0;
                        }
                        else if (isdigit(c))
                           state = 1;
                           else if (c == '.')
                              state = 3;
```

```
            break;
        case 1:
            c = nextchar();
            if (isdigit(c))
               state = 1;
               else if (digit == '.')
                  state=3;
               else state= 2;
            break;
    case 2:
            retract(1); // Backtrack in input
            store_the_lexeme();
            return(INTCON);

        // Cases 3, 4, and 5 would
        // also be implemented here...
```

- Note that since functions can only return one value (and in this case, it is returning just the token), we are storing the actual lexeme somewhere else in memory by calling a function.

8.2.2. Implementing Transition Tables - Arrays

- A two-dimensional array can be used to implement a transition table.
- A table consists of rows representing states and columns representing character classes.
- Examples of character classes include digits / letters (or any punctuation)
- The classes for the previous example (recognizing integer or floating point constants), and an assigned value would be as follows:

Class	Value (column)
Digit	0
.	1
Other	2

- A table representing the transition diagram is as follows:

```
#define ES 6
#define IS -1
int Table[6][3] = {
/* State 0 */ {1, 3, ES},
/* State 1 */ {1, 3, 2},
/* State 2 */ {FS, FS, FS},
/* ... */
/* State ES */ {FS, FS, FS};
```

- Every row in our transition table represents one state, while every column represents a character class.

- **Example**:

  - When in state 1, receiving a character of class 0 (ie. a digit) forces the machine to remain in state 1, a character in class 1 (a '.') forces a transition to state 3, and a character in any other class forces a transition to state 2.

  - ES (Error State) and FS (Final State) are pre-defined constants indicating either an invalid transition (ES) reporting a semantic (spelling) error, or a final state (FS).

- **Note**: The rows and the columns can be swapped in the table.

8.2.3. Implementation of Table-driven Transitions

Given the state table developed previously, the following functions can be used to find the transition from one state to another:

```
int next_state(int currentState, char ch) {
      int column = get_column(ch)
      return Table[currentState][column];'
}


// Given a character, find what class it
// belongs to, to be used for table indexing

int get_column(char ch) {
      if isdigit(ch)
         return(0);
      else if (ch = '.')
         return 1;
      else
         return 2;
}
```

- The NFA implementation is a loop in which a character is taken from the input, the next_state() function is called and the type of the returned state is tested.
- If the state is not-accepting the loop continues.
- If the state is accepting the loop is terminated and the recognized string (lexeme) is processed by the corresponding accepting function.
- Usually, an array of pointers to functions is used to call an accepting function using the accepting state number as an index.

Review Questions

1. Using your words explain what an automaton is.
2. Give examples where automata are used.
3. Why we need to convert NFA into DFA?

Article 9 Transformations: RE to NFA

9.1. Definitions

A **Finite Automaton (FA)** or **Finite State Machine (FSM)** is consists of the following components (parts):

It is proven (**Kleene's Theorem**) that RE and FA are equivalent language definition methods.

- Based on this theoretical result practical algorithms have been developed enabling us actually to construct FA's from RE's and simulate the FA with a computer program using Transition Tables.
- In following this progression:
  - An NFA is constructed first from a regular expression,
  - Then the NFA is reconstructed to a DFA,
  - And finally, a Transition Table is built.
- The **Thompson's Construction Algorithm** is one of the algorithms that can be used to build a Nondeterministic Finite Automaton (NFA) from RE, and Subset construction Algorithm can be applied to convert the NFA into a Deterministic Finite Automaton (DFA). The last step is to generate a transition table (Text § 3.6, 3.7).

We need a finite state machine that is a deterministic finite automaton (**DFA**) so that each state has one unique edge for an input alphabet element. So that for code generation there is no ambiguity.

But a nondeterministic finite automaton (**NFA**) with more than one edge for an input alphabet element is easier to construct using a general algorithm - Thompson's construction. Then following a standard procedure, we convert the NFA to a DFA for coding.

9.1.1. REGULAR EXPRESSION

Example:

- Consider the regular expression:

r = (a|b)*abb

that matches

{abb, aabb, babb, aaabb, bbabb, ababb, aababb,……}

To construct an NFA from this, use **Thompson's construction**:

- This method constructs a regular expression from its components using ε-transitions.
- The ε transitions act as "glue or mortar" for the subcomponent NFA's.
- An ε-transition adds nothing since concatenation with the empty string leaves a regular expression unchanged (concatenation with ε is the identity operation).

Step 1.

Parse the regular expression into its subexpressions involving alphabet symbols a and b and ε:

ε, a, b, a|b, ()*, ab, abb

These describe

a. a regular expression for single characters ε, a, b
b. alternation between a and b representing the union of the sets: L(a) U L(b)
c. Kleene star ()*
d. concatenation of a and b: ab, and also abb

- **Subexpressions** of these kinds have their own Nondeterministic Finite Automata from which the overall NFA is constructed.
- Each component NFA has its own start and end accepting states.

A **Nondeterministic Finite Automata (NFA)** has a transition diagram with possibly more than one edge for a symbol (character of the alphabet) that has a start state and an accepting state. The NFA definitely provides an accepting state for the symbol.

Take these NFA's in turn:

a. the NFA's for single character regular expressions ε, a, b

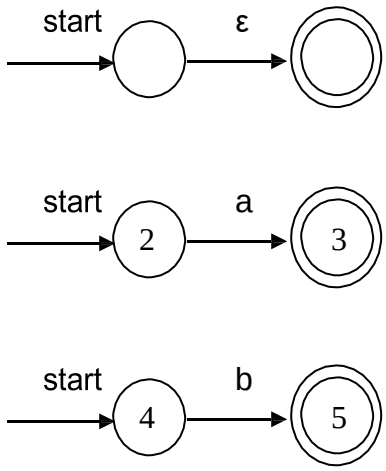


FIG 8.1: NFA Examples

b. the NFA for the union of a and b: a|b is constructed from the individual NFA's using the ε NFA as "glue". Remove the individual accepting states and replace with the overall accepting state

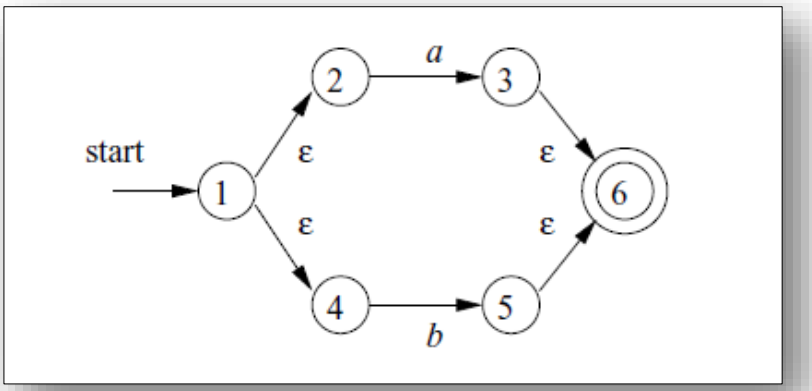


FIG 8.2: NFA using epsilon

c. Kleene star on (a|b)*. The NFA accepts ε in addition to (a|b)*

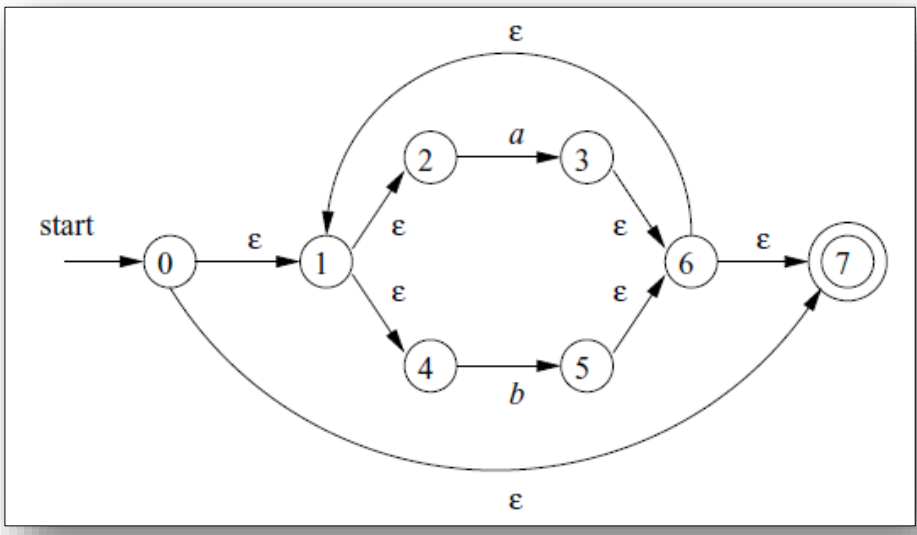


FIG 8.3: NFA composition

d. concatenate with abb

- The problem is that it is not suitable as the basis of a DFA transition table since there are multiple ε, edges leaving many states (0, 1, 6).

Review Questions

1. What is the importance of Thompson Algorithm?
2. Indicate cases where RE are preferable than TD and vice-versa.

Article 10 Transformations: NFA into DFA

10.1. Conversions

10.1.1. Converting the NFA into a DFA

Case I: NFA without ε-transitions.

Sometimes, especially if we had built an NFA without ε-transitions, the process for converting an NFA into a DFA can be simple.

Example:

Let us suppose the following automata.

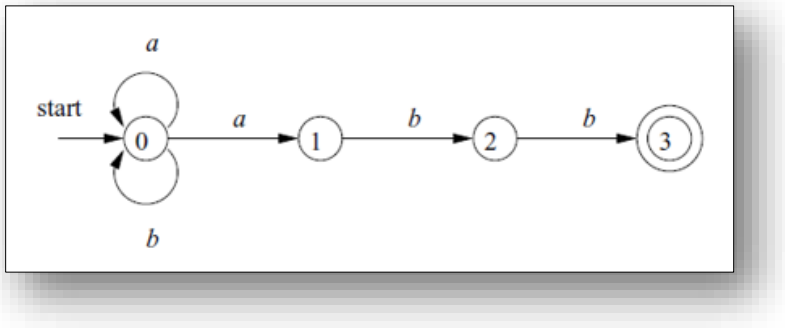


FIG 9.1: Initial automaton to numbers

a. Create the TT (Transition Table):

Q \ Σ	a	b
→ 0	0,1	0
1	-	2
2	-	3
• 3	-	-

b. Now, for each indeterminism, create a new "combined" state and check the output for each symbol of the alphabet:

Q \ Σ	a	b
→ 0	0,1 (=4)	0
1	-	2
2	-	3
• 3	-	-
4	0,1 (=4)	0,2 (=5)

c. Repeat this process successively until you have found all new states:

Q \ Σ	a	b
→ 0	0,1 (=4)	0
1	-	2
2	-	3
• 3	-	-
4	0,1 (=4)	0,2 (=5)
5	0,1 (=4)	0,3 (=6)

Q \ Σ	a	b
→ 0	0,1 (=4)	0
1	-	2
2	-	3
• 3	-	-
4	0,1 (=4)	0,2 (=5)
5	0,1 (=4)	0,3 (=6)
• 6	0,1 (=4)	0

d. Notes about this transformation:
- The initial state remains the same.
- All new states that contain one of the final states, should also be final.
- Now, you can redraw the automaton:

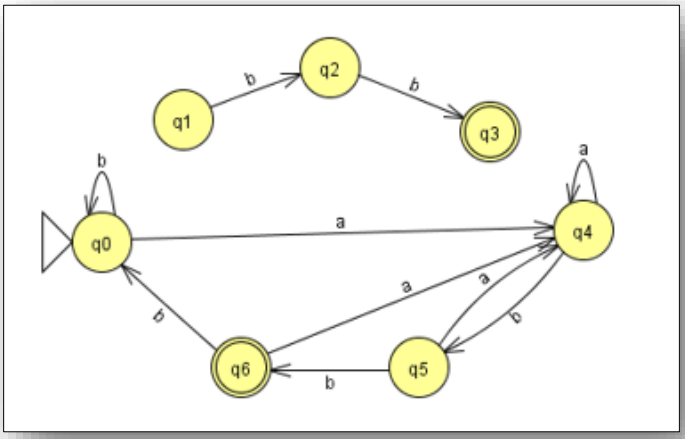


FIG 9.2: JFLAP automaton without optimization

e. Organizing the DFA: Note that, according to the table (and explicitly visible in the diagram, states 1, 2 and 3 are no more accessible from state 0 (initial state). So, the table and the

diagram can be simplified to a new solution.

Q \ Σ	a	b
→ 0	0,1 (=4)	0
4	0,1 (=4)	0,2 (=5)
5	0,1 (=4)	0,3 (=6)
• 6	0,1 (=4)	0

Q \ Σ	a	b
→ 0	4	0
4	4	5
5	4	6
• 6	4	0

f. Finally, it is possible to see the final automaton.

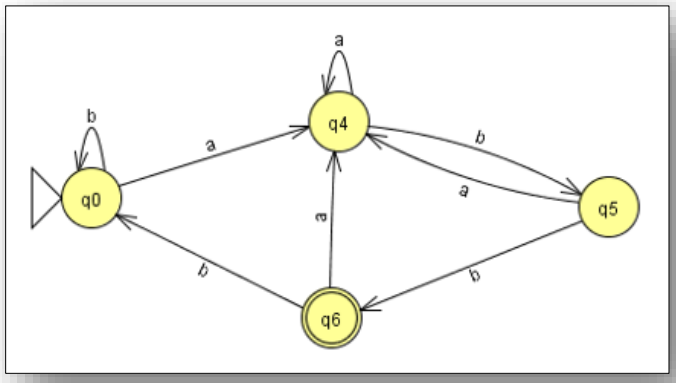


FIG 9.3: JFLAP automaton optimized

Case II: NFA with ε-transitions.

A Deterministic Finite Automaton (DFA) has at most one edge from each state for a given symbol and is a suitable basis for a transition table. We need to eliminate the ε-transitions by subset construction.

Definitions

Consider a single state s. Consider a set of states T:

Operation	Description
ε-closure(s)	Set of NFA states reachable from NFA state **s** on ε-transitions alone
ε-closure(T)	Set of NFA states reachable from set of states **T** on ε-transitions alone
move(T,a)	Set of states to which there is a transition on input symbol **a** from some NFA state in **T**

Let us repeat the previous TD (Transition Diagram) generated by Thompson Algorithm, but doing some marks in states that can start with ε-transitions:

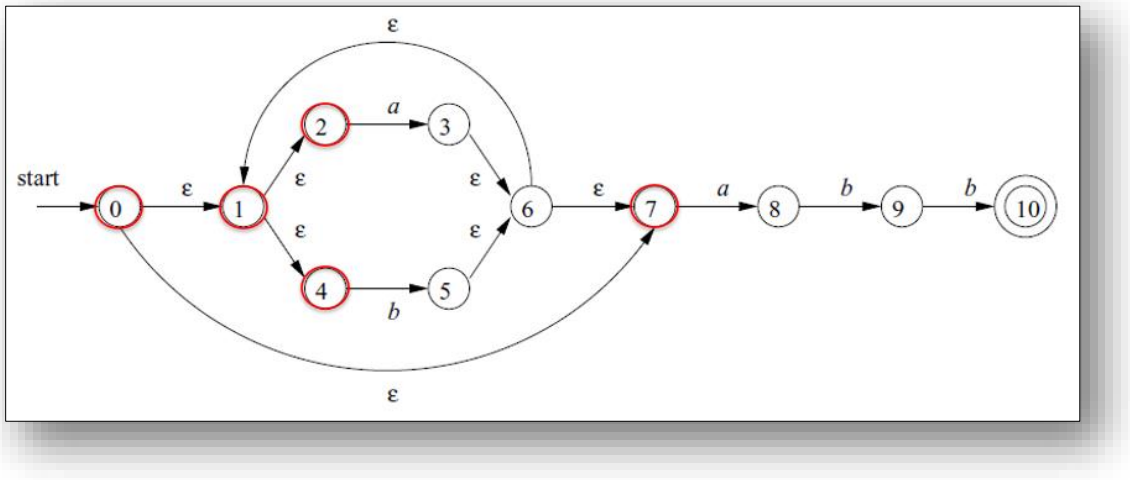


FIG 9.4: ε-transitions from state S0

1. Begin with the start state 0 and calculate ε-closure(0). The set of states reachable by ε-transitions which includes 0 itself is {0,1,2,4,7}. This defines a new state A in the DFA

   A = {0,1,2,4,7}

2. We must find the states that A connects to. Consider A on **a** and from A on **b**. Call these states B and C:

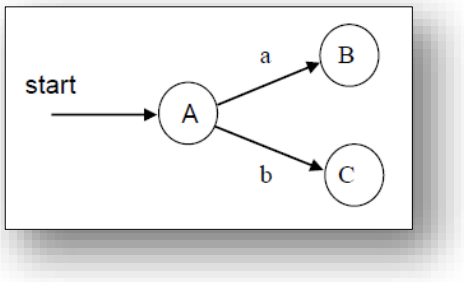


FIG 9.5: Simplification for multiple states

3. Find the state B that has an edge on a from A:

   move(A,a) = {3,8}

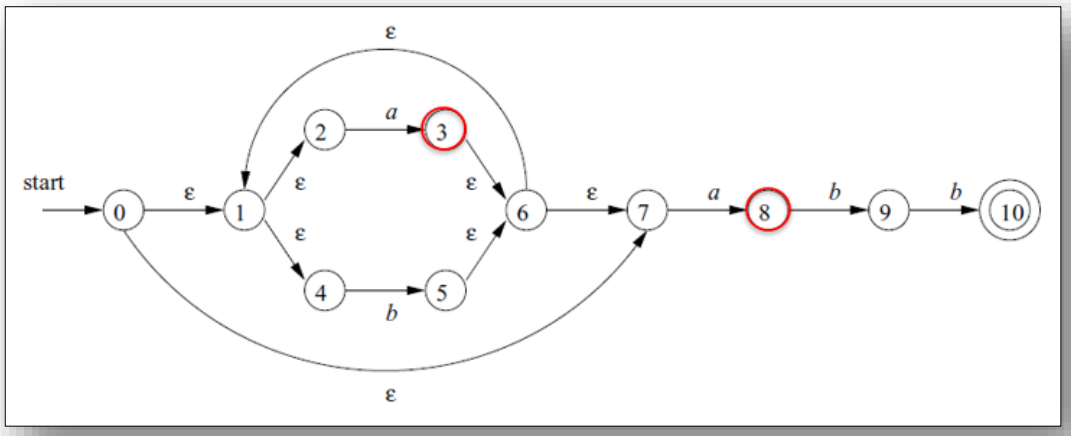


FIG 9.6: NFA composition (intermediate step)

Now check the ε-closure on move:

ε-closure(move(A,a)) = B = {1,2,3,4,6,7,8}

4. Finding the state C with edge from A:

move(A,b) = {5}

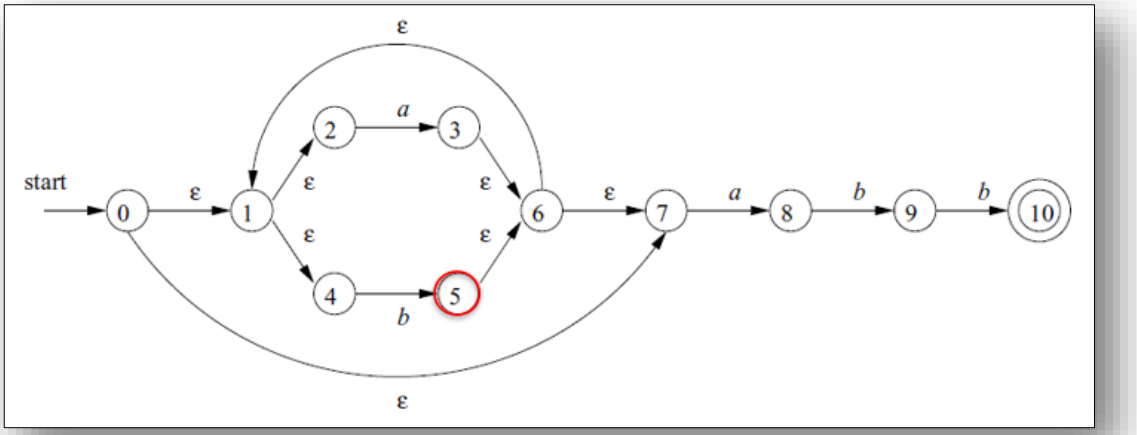


FIG 9.7: NFA composition (intermediate step)

Now check the ε-closure on move(A,b):

ε-closure(move(A,b)) = C = {1,2,4,5,6,7}

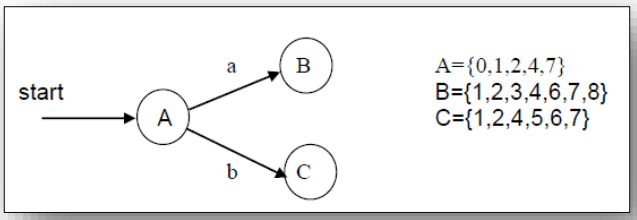


FIG 9.8: Generic representation for new states

5. Find state that has an edge on a from B:

move(B,a) = {3,8}

Seeing ε-closure on move(B,a):

ε-closure(move(B,a)) = {1,2,3,4,6,7,8}

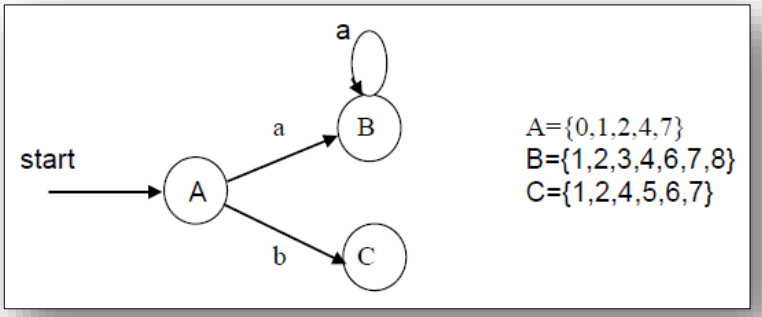


FIG 9.9: Generic representation for new states

6. Finding the state D with edge on b from B:

move(B,b) = {5,9}

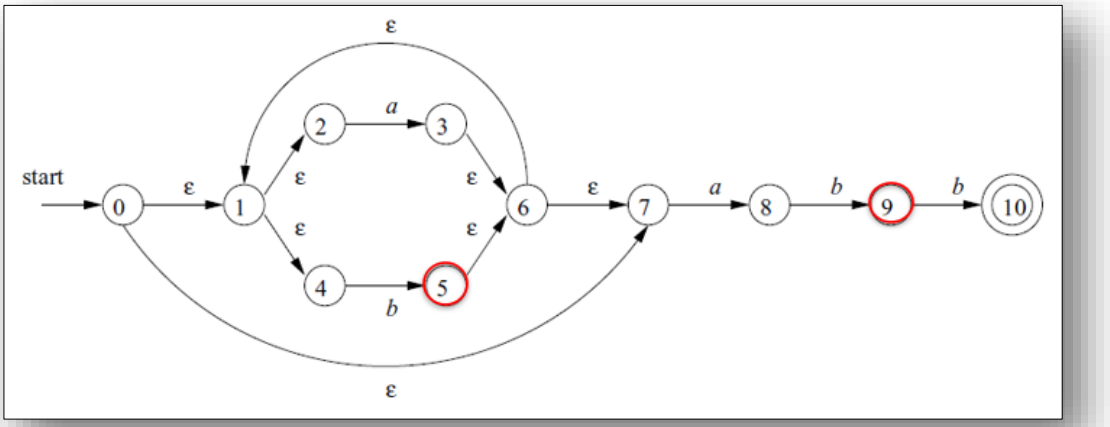


FIG 9.10: NFA composition (intermediate step)

Now check the ε-closure on move:

ε-closure(move(B,b)) = D = {1,2,4,5,6,7,9}

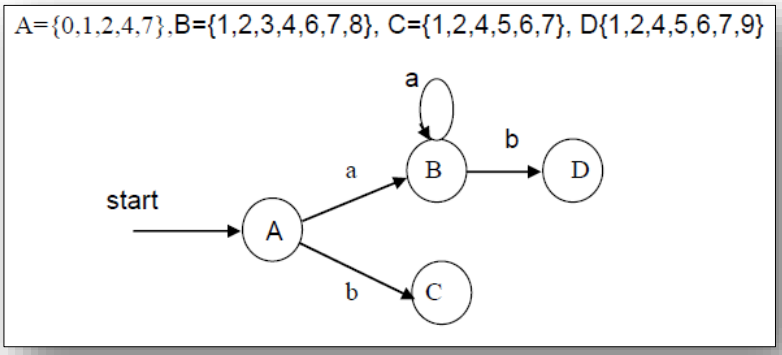


FIG 9.11: New states representation

7. Find state that has an edge on a from D:

move(D,a) = {3,8}

Seeing ε-closure on move(D,a):

ε-closure(move(D,a)) = {1,2,3,4,6,7,8} = B

8. Finding the state E with edge on b from B:

move(D,b) = {5,10}

Now check the ε-closure on move:

ε-closure(move(D,b)) = E = {1,2,4,5,6,7,10}

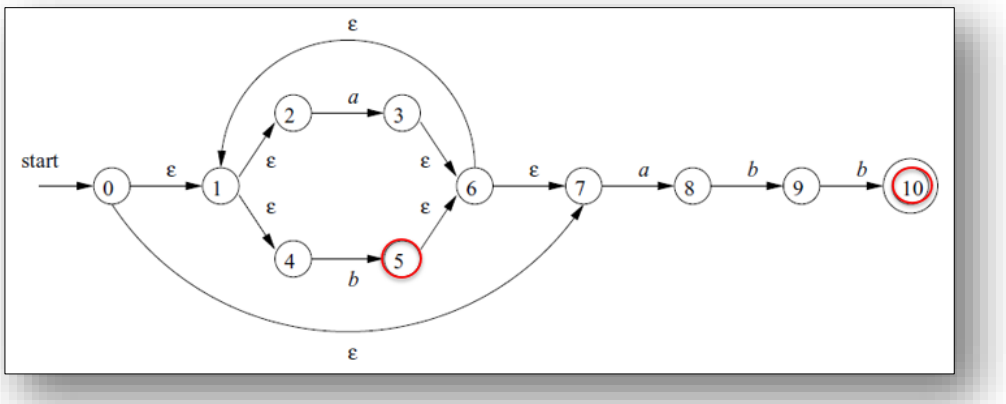


FIG 9.12: NFA composition (intermediate step)

Which can be represented as:

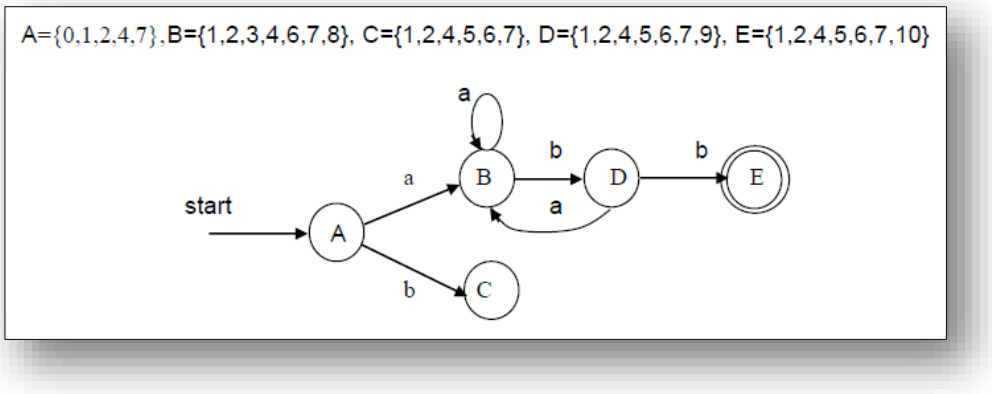


FIG 9.13: New states representation

9. Find state that has an edge on a from C:

move(C,a) = {3,8}

Seeing ε-closure on move(D,a):

ε-closure(move(C,a)) = {1,2,3,4,6,7,8} = B

Represented by:

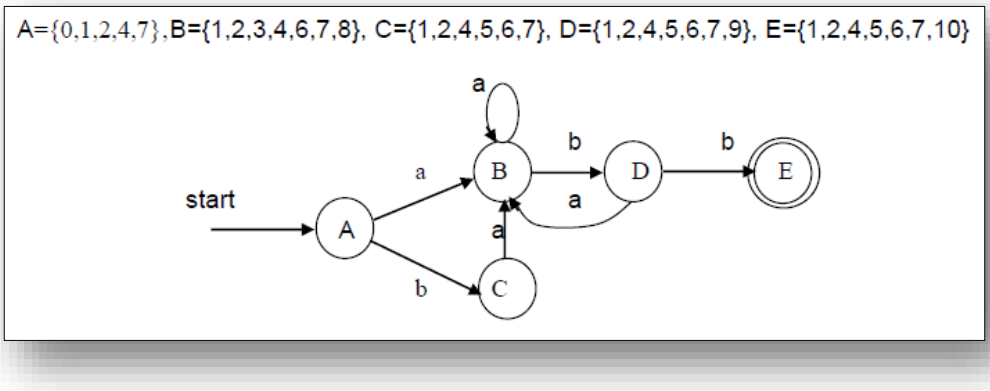


FIG 9.14: New states representation

10. Finding the state with edge on b from C:

move(C,b) = {5}

Now check the ε-closure on move:

ε-closure(move(C,b)) = C

Representation:

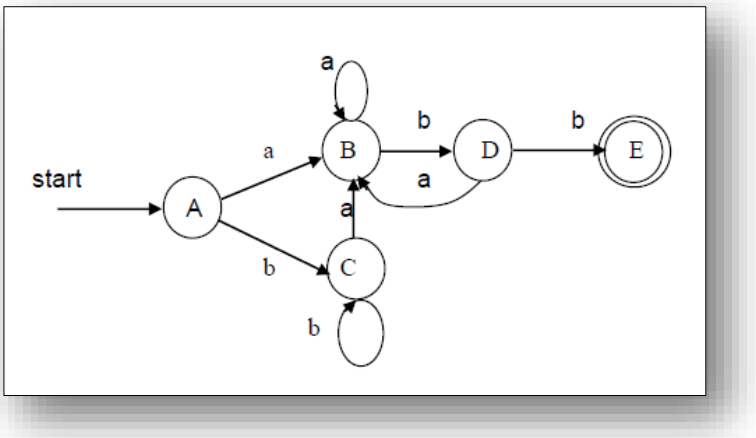


FIG 9.15: New states representation

11. Find state that has an edge on a from E:

move(E,a) = {3,8}

Seeing ε-closure on move(D,a):

ε-closure(move(E,a)) = B

The new representation is:

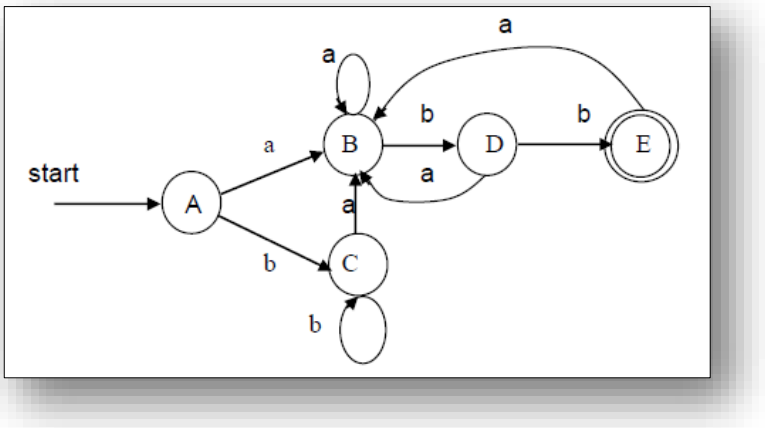


FIG 9.16: New states representation

12. Finally, find the state with edge on b from E:

move(E,b) = {5}

Now check the ε-closure on move:

ε-closure(move(E,b)) = C

Which gives us these final representation:

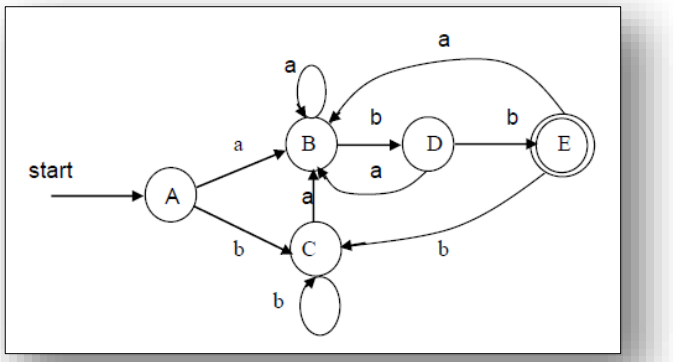


FIG 9.17: New states representation

Now, it is possible to identify the TT (Transition Table). In fact, we have as input the set of N states. We generate as output a set of D states in a DFA.

State	nput a	Input b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

- **Note**: Theoretically an NFA with n states can generate a DFA with 2n states.

10.1.2. NFA vs. DFA (Time-Space Tradeoffs

The table below summarizes the worst-case for determining whether an input string x belongs to the language denoted by a regular expression r using recognizers constructed from NFA and DFA.

Automaton	Space	Time
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

- $|r|$ is the length of ***r***, and $|x|$ is the length of ***x.***

Example:

For the regular expression (a | b)*a(a | b)…(a | b), where there are **n-1** (a | b)'s at the end, there is no DFA with fewer than 2^n states.

Review Questions

1. How you transform an NFA into DFA?
2. What is advantage x advantage of NFA and DFA (in terms of computation: space and time)?

Article 11

Grammar Notations

11.1. Definitions

Formal grammar (represented by "G") is a set of rules.

- It is used to identify **correct** or **incorrect strings** of tokens in a language.
- It is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- It is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Remember Grammar Notations: **$G = (V_T, V_N, P, S)$**, where:

- **V_T**: Set of Terminals.
- **V_N**: Set of Non-Terminals.
- **P**: Productions where $V_N \rightarrow V_N \cup V_T$.
- **S**: Start / Goal symbol.

11.2. Grammar Definitions

- **BNF** stands for Backus-Naur Form.
  - It is used to write a formal representation of a context-free grammar.
  - It is also used to describe the syntax of a programming language.
  - It is basically just a variant of a context-free grammar.
- In BNF, productions have the form:

<Left side> → <definition>

- **Context free grammar:** A formal grammar which is used to generate all possible strings in a given formal language.
- There are the various **applications** of CFG:
  - It is useful to describe most of the programming languages.
  - It helps to properly designed an efficient automatic parser.
  - It is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's, and so on.

Example:

$L = \{wcw^R \mid w \in (a, b)^*\}$

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ **abbcbba**

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

11.3. Derivations

- **Derivation** is a sequence of production rules.
  - It is used to get the input string through these production rules. During parsing we have to take two decisions.
  - These are as follows:
    - We have to decide the non-terminal to be replaced.
    - We have to decide the production rule to be replaced.
  - We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

- **Rule**: the input is scanned and replaced with the production rule from left to right.
  - So, in left most derivatives we read the input string from left to right.
  - Example:

Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

- **Rule**: the input is scanned and replaced with the production rule from right to left.
  - So, in right most derivatives we read the input string from right to left.
  - Example:

Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c
4.

Input:

a - b + c

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

11.4. Notations

The following notation will be used when discussing grammars.

- a, b, c – a **small letter at the beginning of the alphabet** will denote a **terminal**.
  - Note: {a, b , c, …} $\in V_T$.
- A, B, C – a **capital letter at the beginning of the alphabet** will denote a **nonterminal**.
  - Note: {A, B , C, …} $\in V_N$
- X, Y, Z – a **capital letter at the end of the alphabet** will denote a terminal or a **nonterminal**.
  - Note: {…, X, Y , Z} $\in V_T \cup V_N$
- α, β, χ – a **small Greek letter at the beginning of the alphabet** will denote a **string** containing a combination of **terminals and nonteminals**.
  - Note: {α, β, χ, …}: sentential forms of the grammar
- u, v, w – a **small letter at the end of the alphabet** will denote a **string** containing only a

combination of **terminals**.

- Note: {… u, v, w }: sentences of the language defined by the grammar.

Example:

Using the notations described above the following grammar defining arithmetic expressions

```
<expression> →      <expression> + <term>
                  | <expression> - <term> | <term>
<term> → <term> * <factor>
                  | <term> / <factor>
                  | <factor>
<factor> → vid
                  | dil
                  | fpl
                  | (<expression>)
```

This will look like:

```
E → E + T | E – T | T
T → T * F | T / F | F
F → i | d | f | (E) or F → vid | dil | fpl | (E) or
E → E α1 | E α2 | β1
T → T α3 | T α4 | β2
F → i | d | f | (c
```

Review Questions

1. How to represent some productions in your language using the Grammar notation here represented.
2. Can you imagine what is the idea of using Greek letters?

Article 12 Symbol Tables

12.1. Definitions

Symbol table is the structure used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc.

- It is used by both the analysis and synthesis phases.
- The symbol table used for following purposes:
  - To store the name of all entities in a structured form at one place.
  - To verify if a variable has been declared.
  - To determine the scope of a name.
  - To implement type checking by verifying assignments and expressions in the source code are semantically correct.

In short, the symbol table is an association between attributes (meaning = semantics) and names. It is also called dictionary.

Types of Symbol Table (Book – sec. 2.7):

- **Per Scope**: Consider the blocks (TIP: identify vars using the scope – Ex: id_global, id_main, id_func1, id_loop1;

  block → '{' decls stmts '}'

  - **Optimization**: Most-closely nested rule.

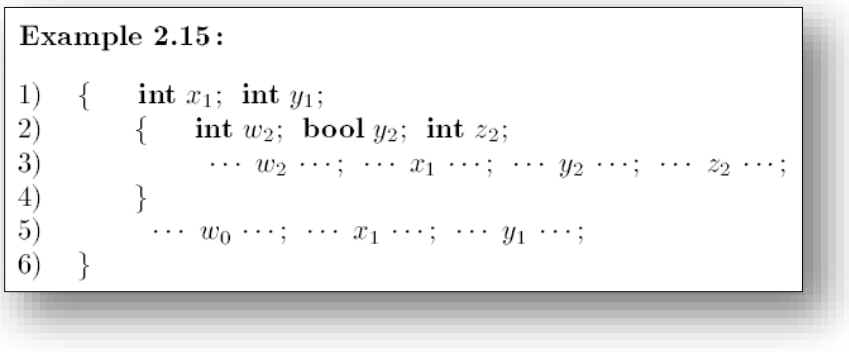


FIG 12.1: Scope representation

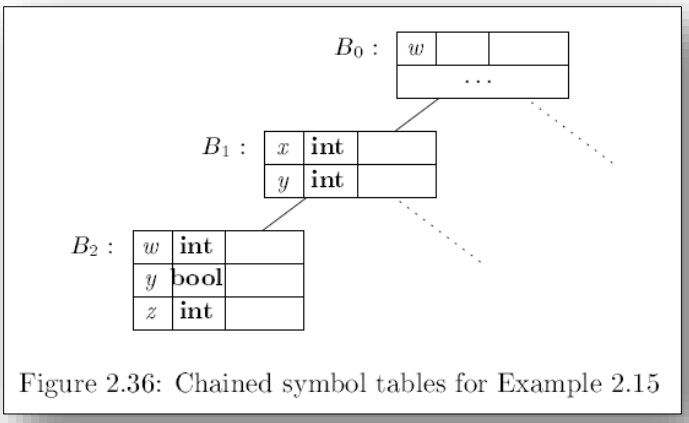


Figure 2.36: Chained symbol tables for Example 2.15

FIG 12.2: Table chain

- **By ordered list:** The trade-off between order / unordered structures:
  - Better way to access (read): ordered;
  - Better way to include (write): unordered.

- **By hash:** If there is space, the usage of Hash tables can improve in both ways (read/write)
  - Common speed: O(1) = Immediate!

```
1) package symbols;                    // File Env.java
2) import java.util.*; import lexer.*; import inter.*;
3) public class Env {
4)    private Hashtable table;
5)    protected Env prev;
6)    public Env(Env n) { table = new Hashtable(); prev = n; }
7)    public void put(Token w, Id i) { table.put(w, i); }
8)    public Id get(Token w) {
9)       for( Env e = this; e != null; e = e.prev ) {
10)         Id found = (Id)(e.table.get(w));
11)         if( found != null ) return found;
12)      }
13)      return null;
14)   }
15) }
```

FIG 12.3: Hashing implementation (Java like code)

12.2. Implementations

Some point of views about Symbol Tables:

- From **theoretical** point of view, a **symbol table** is a mechanism that associates attributes with names. Because these attributes are a representation of the meaning (or semantics) of the names with which they are associated, a symbol table is sometimes called dictionary.
  - A symbol table is a necessary component of a compiler because the introduction of a name appears in only one place in a program, its declaration or definition, whereas the name may be used in any number of places within the program text.
  - Each time the name is encountered, the symbol table provides access to the information collected about the name during the compilation process.
- From **implementation** point of view, a symbol table is a specialized database containing records for names and associated attribute, usually one record per name.
  - A database consists of two parts: Database Manager (Symbol Table Manager) and Database Record Structure.
  - The STM provides services to the client and separates and hides the particular implementation of the database record structure from the client.

Typical **Symbol Table Manager** services are (see the correspondence with **SQL** – Structured Query Language):

- **Create** a symbol table record structure (CREATE)
- Install or **insert** a name record (INSERT)
- **Find** or lookup for a name record (SELECT)
- **Update** a record (UPDATE)
- **Delete** a record (DELETE)
- **Destroy** the symbol table (DROP)
- Other auxiliary services / routines: sort(), print(), pack().

Typical implementation of the symbol table record structure is:

- **Unordered or ordered linear structures**: arrays and linked lists.
- **Hierarchical structures**: binary trees and hash tables.
- **Combinations** of linear and hierarchical structures.

The design choice of a specific implementation depends mainly on the characteristics of the source language the compiler has to translate and to some extend on the compiler implementation language.

- In all cases, the two important leading criteria determining the design of the symbol table are space and speed. The speed is usually determined by the time require for two frequently used operations: adding a record and finding a record.

Review Questions

1. Summarize the importance of Symbol Table (ST).
2. How is it updated?
3. How do you implement a ST?
4. What are the corresponding SQL routines?

Article 13

Parsing Overview

13.1. Overview

The **Parser** analyzes streams of tokens retrieved from the lexical analyzer, to see if they produce a valid string in the language.

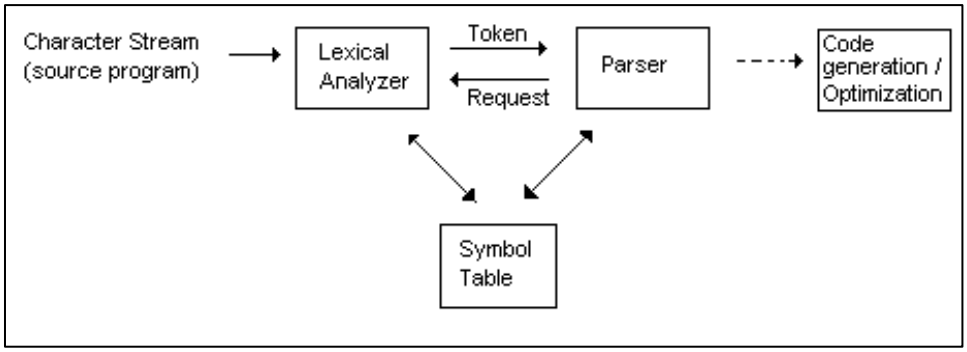


FIG 13.1: Review of compilation process

- The parser also provides information to be used by later stages in the compiler
- The symbol table may be used and updated
    - For example, attributes may be set in the symbol table, such as whether an identifier is a function name or a variable.

13.1.1. LL vs. LR Grammars

Grammars are divided into **several classes**.

Note that any programming language can typically be described using grammars of either type. (There are also algorithms which can change one grammar to another.) Two Grammars are equivalent if they can generate the same language (although with different parse trees)

- **LL Grammars** - Input to the parser is accepted starting from the Left (first L), and the left-most derivative is taken (second L)
- **LR Grammars** - Input to the parser is accepted starting from the Left (the L), and the right-most derivative is taken (the R)

Often, a number is attached to the grammar, to indicate the minimum number of tokens the parser needs to look ahead in the input stream.

- Example: LL(1) grammars look ahead by one token.
- If no number is given, it is assumed to be one.

13.1.2. Specifying Derivations (notation)

There are several ways to view the process by which a grammar defines sentences of a language.

- Up to now we viewed this process as one of building parse trees.
- But it can be also defined by linear process of nonterminal replacement called derivations.
- The central idea behind derivations is that a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production.
- When a sequence of derivations is made, the application of a production can be specified using a $\rightarrow$ symbol.
- If the derivation is left-most, it will often have an 'lm' below the $\rightarrow$ symbol

Example:

$$\text{Expr} \underset{lm}{\rightarrow} \text{Expr} + \text{Term} \underset{lm}{\rightarrow} \text{Term} + \text{Term}$$

- We can also use an * above the $\rightarrow$ symbol to indicate a string can be derived (perhaps in multiple steps) from the non- terminal in the right-hand side
- If left derivations are used to parse the input string, the non- terminal is said to be in left-sentinel form.
- If right derivations are used to parse the token stream, they are sometimes called canonical derivations.

13.1.3. Top Down parsing

The parser starts with the start symbol and decides which productions can be used to give the token stream.

- Lower-level productions are subsequently tried, in an attempt to find which productions, match the input
- Substituting non-terminals for their productions is done on a left to right basis.
- Sometimes, a production may be tried, but it is later found that it is unsuitable for giving the desired token stream... In this situation, the parser may have to backtrack.
- Best for use with LL grammars.
- Most suitable method for writing by hand.

13.1.4. Top Down Parsing - Example

- Given a Grammar for expressions:

Expr $\rightarrow$ Expr + Term | Expr - Term | Term Term $\rightarrow$ Num | Var

And the input stream:

x + 1 (This is equivalent to Var+Num)

The following steps result:

```
Expr → Expr + Term
     lm
      → Term + Term
     lm
      → Var + Term
     lm
      → Var + Num
     lm
```

13.1.5. Parse Tree for Top Down Parsing

The parse tree for the previous example would begin with just the start symbol:

```
Expr
```

- The expression would be broken down:

```
      Expr
    /   |   \
Expr    +    Term
```

- The non-terminal Expr (ie. the ***left-most***) would be broken down next:

```
      Expr
    /   |   \
Expr    +    Term
  |
Term
```

- In the last two trees, the last two terminals are resolved.

```
      Expr
    /   |   \
Expr    +    Term
  |
Term
  |
 Var
```

And:

```
          Expr
        /   |   \
     Expr   +   Term
      |           |
     Term        Num
      |
     Var
```

13.1.1. Other examples

New example: Given grammar is G2:

```
S -> Ac
A -> ab
```

where S is start symbol, A is non-terminal and a, b, c are terminals.

- Input string: abc
- Parse tree generated by LL parser:

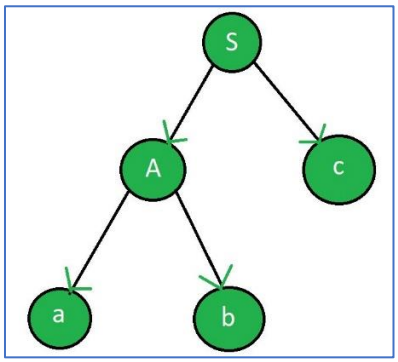


FIG 13.2: Top-down parse tree

Given the following grammar (G3):

```
S → cAd
A → aXb | X
X → e | f
```

- If we are given the string: cafbd, the derivations to build this string are as follows:

```
S → cAd → caXbd → cafbd
```

13.1.2. Bottom Up Parsing

The parser tries to match the low-level productions first, gradually working up towards building high level productions

- Suited for LR Grammars
- Also suited for automated implementation;
- Many tools exist which create bottom-up parsers, such as: YACC (https://invisible-island.net/byacc/), Bison (https://www.gnu.org/software/bison/), JavaCC (https://javacc.github.io/javacc/).

Shift reduce strategy:

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- It uses a stack to hold the grammar and an input tape to hold the string.
- It performs the two actions: **shift** and **reduce**.
    - At the **shift** action, the current symbol in the input string is pushed to a stack.
    - At **reduction** moment, the symbols will be replaced by the non-terminals.
        - The symbol is the right side of the production and non-terminal is the left side of the production.

Example:

- Given the same grammar as used in bottom-up parsing

```
Expr → Expr + Term | Expr - Term | Term
Term → Num | Var
```

- And the same input stream:

x + 1 (Basically, a Var + Num)

- Parsing begins with the first token:

```
Var
```

- A production is applied to obtain the following:

```
Term
  |
Var
```

- After several more tokens are analyzed, and additional productions are applied, the following is produced:

```
Expr + Term
  |      |
Term   Num
 |
Var
```

- Finally, the completed tree is produced:

```
      Expr
    /   |   \
Expr  +   Term
  |         |
Term       Num
  |
 Var
```

Another (more theoretical) Example

- Given the following grammar:

```
S → aABe
A → Abc | b
B → d
```

- If we are given the string: abbcde, the right-mos derivations to build this string are as follows:

```
S → aABe → aAde → aAbcde → abbcde
    rm      rm        rm          rm
```

- The sentence abbcde can be reduced to S by the following steps:

```
Abbcde
aAbcde
aAde
aABe
S
```

These reductions, in fact, trace out the right-most derivation in reverse.

- There are many bottom-up parsers: **LR(0), SLR(1), LR(1), LR(k), LALR.**

Operator-precedence parsers: They all use parsing tables to perform the shift-reduce parsing

operations.

New example: Review the following grammar (G2):

```
S -> Ac
A -> ab
```

- Parse tree generated by LR parser:

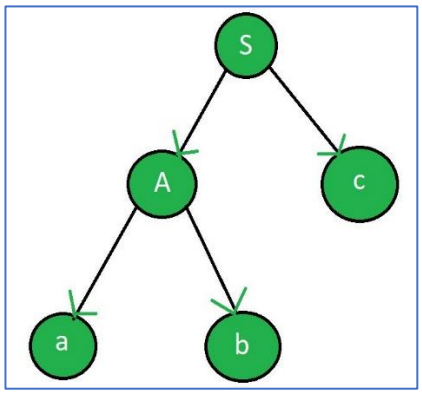


FIG 13.3: Button-up parse tree

In short:

- **LL Parser** includes both the recursive descent parser and non-recursive descent parser. It is one type uses backtracking while another one uses parsing table.
- **LR Parser** is one of the bottom up parser which uses parsing table (*dynamic programming*) to obtain the parse tree form given string using grammar productions.

Review Questions

1. Define the functionality of top-down parsing.
2. Describe one strategy of error recovering.
3. How LL / LR parsers differ?

Article 14 Grammar and Parsing

14.1. Derivations

We look at a description of a grammar as a way of deriving a language and how to design a grammar suitable for a recursive descent parser:

- Our objective is a recursive descent parser where the parse tree is not actually generated but is implicit in the derivations
- A recursive descent parser can only work for certain types of grammars with the right kind of productions.

The derivation view is that the sentences of the language are generated by repeated application of the productions.

In general, a string α can consist of a mixture of terminals and nonterminals.

A production that derives the string α from the nonterminal A has the form:

$$A \rightarrow \alpha$$

Since strings are derived from other strings, a series of productions that generates αn from α_1 is written:

$$\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \quad \ldots \rightarrow \quad \alpha_N$$

- The symbol $\rightarrow$ means “generates in one step”
- To summarize the above equation, we can use the symbol $\rightarrow^*$ meaning “generates in 0 or more steps”. Thus,

$$\alpha_1 \rightarrow^* \alpha_N$$

In general, a string α is in sentential form if it can be generated by the grammar:

$$S \rightarrow^* \alpha$$

If a string ω contains only terminals, then it must have been derived from the start symbol by at least one production. We write this as:

$$S \rightarrow^+ \omega$$

This equation defines the **grammar**. Such a string is a sentence in the **language**.

14.1.1. The Process

a. Derives in one step.

The meta symbol for this is =>.

- We say a string v **derives** a string w if a production can be used to produce w from v. So

$$v \rightarrow w$$

- if we can write for some strings v and w (where x and z are other strings of terminals and non-terminals)

$$v = xYz, \qquad w = xyz$$

- where $Y \rightarrow y$ is a production in the grammar that produces the string y from the nonterminal Y.
- If the strings x and v are empty, then we have

$$Y \rightarrow y$$

b. Derives in one or more steps.

The meta symbol for this is $\rightarrow^+$

- We say a string v derives a string w in one or more steps if w is produced from v after applying one or many productions:

$$v \rightarrow w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_n = w$$

summarized as:

$$v \rightarrow^+ w$$

c. Derives in zero or more steps

The metasymbol for this is $\rightarrow$

- This is =>+ together with no derivation:

$$v \rightarrow^* w \text{ if } v \rightarrow^+ w \text{ or } v = w$$

Finally, we arrive at another way of thinking of a language: Let **$G[V_T, V_N, P, S]$** be a grammar (S is the start symbol).

- A string v is called a sentential form if v is derivable from S:

$$S \rightarrow^* v$$

- A string is a sentence if v consists only of terminals $v = V_T^+$.
- The language $L(G[V_T,V_N,P,S])$ is therefore the set of sentences produced by the grammar:

$$L(G[V_T,V_N,P,S]) = \{v \mid S \rightarrow v \text{ and } v \text{ in } V_T\}$$

A language is the set of terminal strings (sentences) that is produced by the grammar. The sentences are a subset of all possible terminal strings.

14.1.2. Left-most and Right-most Derivations

Consider the following grammar:

```
expr → expr op expr | ( expr ) | - expr | id
op   → + | - | * | /
```

- This first line is productions for **expr**. The second line is the productions for **op**. expr and op are nonterminals. id and +, -, * and / are terminals.

Example:

Consider this example from the grammar. The string –(x + y) is a sentence of the grammar because:

Left-most derivation

```
expr      → -expr → -(expr)
          → -(expr op expr)
          → -(id op expr)
          → -(x op expr)
          → -(x + expr)
          → -(x + id)
          → -(x + y)
```

Or:

```
          expr →* -(x + y)
```

This is left-most because the left-most nonterminal is replaced at each stage.

The alternative is to replace the right-most nonterminal at each stage:

Right-most derivation

```
expr    → -expr
        → -(expr)
        → -(expr op expr)
        → -(expr op id)
        → -( expr op y)
        → -( expr + y)
        → -( id + y)
        → -(x + y)
```

- **Note:** left-most and right-most derivations generate the same parse tree and every parse tree has associated with it a unique left-most and unique right-most derivation.
- A type of parsing algorithm LL(1) that we will meet later takes its name from the fact that it reads the input token string from Left (first L) to Right, does a left-most (second L) derivation, and looks one token ahead to determine the production to use.

14.1.3. Left and Right Associativity and Left and Right Recursion

Consider the following strings: 9 – 5 – 2 and a = b = c

Example 1: 9 – 5 – 2

The – operator is left associative since the 5 associates with the – on its left.

The normal interpretation of the expression is:

(9 – 5) – 2 = 2

not

9 – (5 – 2) = 6 wrong

Example 2: a = b = c

The = operator is right associative since the b associates with the = on its right The normal interpretation of the expression is:

a = (b = c): c is assigned to b, then to a

not

(a = b) = c wrong

Question:

- How can this associativity be represented in grammars? Consider two grammars that implement this associativity:

Option 1: left associativity for strings of digits with arithmetic operators

```
list → list + digit | list – digit | digit
digit → [0-9]
```

- Left associativity is enforced by evaluating left associating expressions lower down in the parse tree where precedence is higher.

Option 2: right associativity for strings of characters with assignment

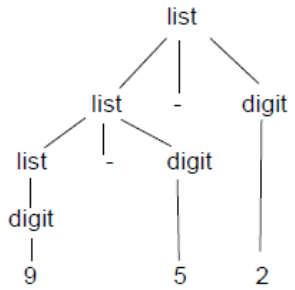
```
right → letter = right | letter
letter → [a-z]
```

- Right associativity is enforced by evaluating right associating expressions lower down in the parse tree where precedence is higher.

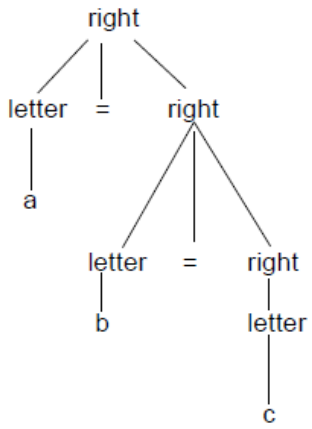
14.2. Parse Trees

Parse trees generated for these expressions are (using a left-most derivation and one derivation at each stage):

Example 1: 9 – 5 – 2



Example 2: a = b = c



Note how the right associative tree grows down to the right.

14.3. Recursion

14.3.1. Left and Right Recursion (Text § 2.4, 4.3)

Left-associative grammars naturally support the associativity of some operators, but there is a serious problem with recursive-descent parsers.

- Recursive descent parsers implement the productions as function calls.
- So, if the nonterminal on the LHS (left hand side) is also the first symbol on the RHS (right hand side), the function will enter infinite recursion.

In general, productions look like:

- **Left Recursion: A → A α | β**
- **Right Recursion: A → α A | β**

where α and β are sequences of terminals and non-terminals that do not begin with A.

- In general a grammar is left-recursive if it has a nonterminal such that there is a derivation:

$$A \rightarrow^* A\ \alpha$$

The strings generated by these productions are of the form:

1. Using left recursion

$$A \rightarrow A\ \alpha \rightarrow A\ \alpha\ \alpha \rightarrow A\ \alpha\ \alpha\ \alpha \rightarrow \ldots\ldots\ldots\ \beta\ \alpha\ \alpha\ \ldots\alpha$$

Set of stings generated by the grammar:

$$\{\ \beta,\ \beta\alpha,\ \beta\alpha\alpha,\ \beta\alpha\alpha\alpha,\ \beta\alpha\alpha\alpha\alpha\ \ldots\ \}$$

Parse tree:

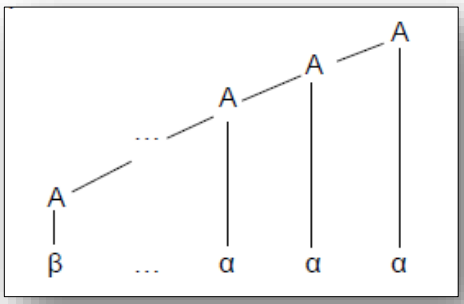


FIG 14.1: Left recursion

2. Using right recursion

$$A \rightarrow \alpha\ A \rightarrow \alpha\ \alpha\ A \rightarrow \alpha\ \alpha\ \alpha\ A \rightarrow \ldots\ldots\ldots\ \alpha\ \alpha\ \ldots\alpha\ \beta$$

Set of stings generated by the grammar:

$$\{\ \beta,\ \alpha\beta,\ \alpha\alpha\beta,\ \alpha\alpha\alpha\beta,\ \alpha\alpha\alpha\alpha\beta\ \ldots\ \}$$

- Note that a recursive-descent parser that attempts to apply the production as a function call would result in the evaluation of the first symbol on RHS which is the same function call, and the beginning of infinite recursion.
- Hence in such a parser, we need to remove left recursion but implement the code so as to preserve left associativity.

14.3.2. Eliminating Left Recursion

We can eliminate a left recursion (that follows naturally from the grammar) into a right recursion with the following algorithm that can be applied to any production and does not change the set of derivable strings.

Original:

$$\mathbf{A \rightarrow A\ \alpha\ |\ \beta}$$

Set of stings generated by the grammar:

$$\{\ \beta,\ \beta\alpha,\ \beta\alpha\alpha,\ \beta\alpha\alpha\alpha,\ \beta\alpha\alpha\alpha\alpha\ \ldots\ \}$$

Converted:

$$\mathbf{A \rightarrow \beta\ A'}$$
$$\mathbf{A' \rightarrow \alpha\ A'\ |\ \varepsilon}$$

- Note that we now have right-recursion and an ε-production, but the strings generated are the same as the left recursive case:

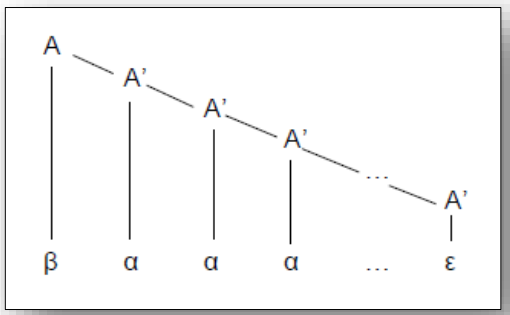


FIG 14.2: Right recursion

The same set of set of stings generated by the modified grammar:

$$\{ \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \beta\alpha\alpha\alpha\alpha \ldots \}$$

Original:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta$$

Strings produced:

$$\{ \beta, \beta\alpha1, \beta\alpha2, \beta\alpha1\alpha2\alpha1, \beta\alpha1\alpha1\alpha1 , \beta\alpha2\alpha1,\alpha2 , \beta\alpha2\alpha2,\alpha2 \ldots \}$$

Converted:

$$A \rightarrow \beta\ A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$$

Strings produced:

$$\{ \beta, \beta\alpha_1, \beta\alpha_2, \beta\alpha_1\alpha_2\alpha_1, \beta\alpha_1\alpha_1\alpha_1 , \beta\alpha_2\alpha_1,\alpha_2 , \beta\alpha_2\alpha_2,\alpha_2 \ldots \}$$

14.3.3. Examples

Take the left-recursive part of the grammar above

```
list      →    list + digit | list – digit | digit
A         →    A α1 | A α2 | β
```

with left recursion removed:

```
list      →    digit list'
list'     →    + digit list' | – digit  list' | ε
```

The general rule is:

```
A  → Aα1 | Aα2 | Aα3 | ... | Aαm| β1| | β2 | …. | βn
A  → β1A' | β2A'| ... | βnA'
A' → α1A' | α2A'| ... | αmA' | ε
```

- **Note**: these are examples of immediate left recursion. The recursion could be several steps later:

```
S → A a | b
A → A c | S d | ε
```

Extra examples:

<expression> → <expression> + <term>
| <expression> - <term> | <term>
<term> → <term> * <factor>
| <term> / <factor> | < factor >
< factor > → vid | dil | fpl | (<expression>)

Rewriting:

E → E + T | E – T | T
T → T * F | T / F | F
F → vid | dil | fpl | (E)

Replacing:

E → E α1 | E α2 | T
T → T β1 | T β2 | F
F → i | d | f | (γ

Where:

- α1 = + T
- α2 = - T
- β1 = * F
- β2 = / F
- γ = E)

14.3.4. Left Factoring a Grammar (Text § 2.4, 4.3)

The grammar

```
relational_expression → primary_expression > primary_expression
                      | primary_expression < primary_expression
```

is not suitable for predictive parsing.

- It is not possible to make the decision which of the alternatives to choose having one token lookahead because all production alternatives have the same left-most terminal.
- It is possible to rework the grammar so that the decision can deferred until the parser can make the right choice.
- The corresponding grammar transformation is called left factoring.

The general algorithm for left-factoring a grammar is shown as follows.

1. Given a grammar

   $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

2. If α is not ε the left factored grammar is:

   $A \rightarrow \alpha A' \mid \gamma$

   $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

   - This transformation must be applied repeatedly until no alternatives for a nonterminal have a common prefix.
   - In many cases the grammar may be transformed without applying the general rule. For example, the grammar above can be rewritten as

```
relational_expression →
     primary_expression relational_operator primary_expression
relational operator →  < | >
```

14.4. First and Follow Set

14.4.1. Building the FIRST and FOLLOW sets for a Grammar

Case 1: FIRST set

The construction of a predictive parser requires two set of tokens to be built.

The **FIRST** set is a set of tokens that appear at the left-most position after zero or more derivations are applied to a grammar production. The formal definition is

$$\textbf{FIRST(A)} = \{ a \mid A \Rightarrow a\alpha \}$$

The following rules for computing the FIRST set can be derived from the formal definition.

- **BFTS1**. If X is a terminal, then **FIRST(X) = {X}**
- **BFTS2**. If X is a nonterminal and X → ε is a production, then add ε to **FIRST(X)**.
- **BFTS3**. If X is a nonterminal and X → $Y_1Y_2\ldots Y_K$ is a production, then:
  - **FIRST(X) = {FIRST(Y_1)}** if Y_1 does not derive **ε;**
  - **FIRST(X) = {FIRST(Y_1), FIRST(Y_2)}** if Y_1 derives **ε;**
  - **FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3)}** if Y_1 and Y_2 derive **ε;**
  - **FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_i)}** if **ε** is in all **FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_{i-1})**;
  - **FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_k), ε }** if ε is in all ;

- **BFTS4**. If X is a nonterminal and X → **A** | **B**… | **D** is a production then **FIRST(X) = {FIRST(A), FIRST(B),…, FIRST(D) }**.

Case 2: FOLLOW set

The FOLLOW set is a set of tokens that appear on the right side of a nonterminal after zero or more derivations a grammar production. The formal definition is

$$\mathbf{FOLLOW(B) = \{\ a \mid A \rightarrow^* \alpha B a \gamma\ \}}$$

The following rules for computing the FOLLOW set can be derived from the formal definition.

- **BFWS0.** The **FOLLOW** set cannot contain **ε**.
- **BFWS1**. If **S** is the start symbol (nonterminal) for a grammar place **$** in the **FOLLOW(S),** where **$** is the input end-marker (for example, end of file)
- **BFWS2**. If there is production A → α**B**a | α**B**b, then **FOLLOW(B) = {a,b}**
- **BFWS3**. If there is production A → α**B**β and β does not derive **ε**, then add **FIRST(β)** to the **FOLLOW(B)** set.
- **BFWS4**. If there is production A → α**B,** then add **FOLLOW(A)** to the **FOLLOW(B)** set
- **BFWS5**. If there is production A → α**B**β and β derives **ε**, then add **FOLLOW(A)** to the **FOLLOW(B)** set.

The following rules for computing the FOLLOW set can be derived from the formal definition.

14.4.2. Examples

Example 1: FIRST set

BFTS1. If X is a terminal, then FIRST(X) = {X}

FIRST(a) = {a}

BFTS2. If X is a nonterminal and X → ε is a production, then add ε to FIRST(X)

A → ε

FIRST(A) → {ε}

BFTS3. If X is a nonterminal and X → $Y_1Y_2…Y_k$ is a production, then FIRST(X) = {FIRST(Y_1)} if Y_1 does not derive ε;

FIRST(X) = {FIRST(Y_1), FIRST(Y_2)} if Y_1 derives ε;

if ε is in all FIRST(Y_1), FIRST(Y_2), FIRST(Y_3),…FIRST(Y_k);

```
A → BCD
B → a
C → c
D → d
A → BCD → aCD
FIRST(A) = {FIRST(B),…} = {a}
```

```
A → BCD
B → a | ε
C → c
C → d
A → BCD → aCD
A → BCD → εCD → CD → cD
FIRST(A) = {FIRST(B),FIRST(C)…} = {a,c}
```

```
A → BCD
B → a | ε
C → c | ε
D → d
A → BCD → aCD
A → BCD → εCD → CD → cD
A → BCD → εCD → CD → εD → D → d
FIRST(A) = {FIRST(B),FIRST(C),FIRST(D),…} = {a,c,d}
```

```
A → BCD
B → a | ε
C → c | ε
D → d | ε
A → BCD → aCD
A → BCD → εCD → CD → cD
A → BCD → εCD → CD → εD → D → d
A → BCD → εCD → CD → εD → D → ε
FIRST(A) = {FIRST(B),FIRST(C),FIRST(D),…} = {a,c,d,ε}
```

BFTS4. If X is a nonterminal and X → A | B… | D is a production, then FIRST(X) = {FIRST(A), FIRST(B),…, FIRST(D) }.

```
A → B | C | D
B → a
C → c
D → d
A → B → a
```

```
A → C → c
A → D → d
FIRST(A) = {FIRST(B),FIRST(C),FIRST(D)} = {a,c,d}
```

Example 2: FOLLOW set

BFWS1. If S is the start symbol (nonterminal) for a grammar place $ in the **FOLLOW(S),** where **$** is the input end-marker (for example, end of file)

```
A → BCD
FOLLOW(A) = {$}
```

BFWS2. If there is production A → αBa | αBb, then FOLLOW(B) = {a,b}

```
A → αBa | αBb
FOLLOW(B) = {a,b}
```

BFWS3. If there is production A → αBβ and β does not derive ε, then add FIRST(β) to the FOLLOW(B) set.

```
A → BCD B → b
C → c D → d
FIRST(C) = {c}
FIRST(D) = {d}

FOLLOW(A) = {$}
FOLLOW(B) = {FIRST(C)} = {c}
FOLLOW(C) = {FIRST(D)} = {d}
FOLLOW(C) = {FIRST(D)} = {d}
```

BFWS4. If there is production A → αB, then add FOLLOW(A) to the FOLLOW(B) set.

```
FOLLOW(D) = {FOLLOW(A)} = {$}
```

BFWS5. If there is production A → αBβ and β derives ε, then add FOLLOW(A) to the FOLLOW(B) set.

```
A → BCD
B → b
C → c | ε
D → d | ε
FIRST(C) = {c, ε}
FIRST(D) = {d, ε}

FOLLOW(A) = {$}
FOLLOW(B)= {FIRST(C),…} = {c,…} = {c, FIRST(D)…} = {c,d,…} =
{c,d,FOLLOW(A)} = {c,d,$}
```

Example 3: Grammar Modification

In this final example we will use the simple **Math Expression** to create FIRST and FOLLOW sets after the modification.

Original Grammar:

```
E → E + T | T
T → T * F | F
F → v | ( E )
```

Transformed Grammar:

```
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → v | (E)
```

FIRST set:

```
FIRST(E) = {FIRST(T),…} = {v,(}
FIRST(E') = {+,ε}
FIRST(T) = {FIRST(F),…} = {v,(}
FIRST(T') = {*,ε}
FIRST(F) = {v,(}
```

FOLLOW set:

```
FOLLOW(E) = {$,)}
FOLLOW(E') = {FOLLOW(E),FOLLOW(E')}
             = {$,),$,)}
             = {$,)}
FOLLOW(T) = {FIRST(E'),FIRST(E')}
             = {+,FOLLOW(E},+,FOLLOW(E')}
             = {+,$,),+,$,)}
             = {+,$,)}
FOLLOW(T') = {FOLLOW(T),FOLLOW(T')}
             = { +,$,),+,$,)}
             = {+,$,)}
FOLLOW(F) = {FIRST(T'),FIRST(T')}
             = {*,FOLLOW(T'},*,FOLLOW(T')}
             = {*,+,$,),*,+,$,)}
             = {*,+,$,)}
```

14.5. Error Handling and Sets

1. As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A.
    - If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
2. It is **not enough** to use FOLLOW(A) as the synchronizing set for A.
    - For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal expressions.
    - A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.
    - Often, there is a hierarchical structure on constructs in a language.
    - For example, expressions appear within statements, which appear within blocks, etc.
    - We can add to the synchronizing set of a lower-level construct the symbols that begin higher-level constructs.
    - For example, we might add keywords that begin statements to the synchronizing sets for the non-terminals generating expressions.
3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to **resume parsing** according to A if a symbol in FIRST(A) appears in the input.
4. If a nonterminal can generate the **empty string**, then the production deriving e can be used as a default.
    - Doing so may **postpone some error** detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
5. If a terminal on top of the stack cannot be matched, a simple idea is to **pop the terminal**, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

Figure 4.22: Synchronizing tokens added to the parsing table of Fig. 4.17

FIG 14.3: Parsing table with synchronization

Review Questions

1. What is the importance of solving left recursion and left transformation for unpredictability?
2. Explain what the definition and importance of and FIRST FOLLOW is set.

Article 15 Non-Recursive Predictive Parsing

15.1. General View

In **recursive descent parsing**, a nonterminal is implemented as a function call and the right-hand side of its production as the body of the function.

- Since the body may consist of other nonterminal, then it will call other functions. The **system stack** is used to pass parameters to and from the functions.
- Instead of using the system stack, it is possible for the parser to maintain its own stack and integrate it more closely with the parsing process.
- In addition, since the main task of the function calls is to match the **lookahead token**, matching can now be done explicitly using grammar symbols (representing productions) on the stack.
- Instead of recursive function calls, a table is now used (analogous to the lexical transition table) to guide the parser to a matching token.

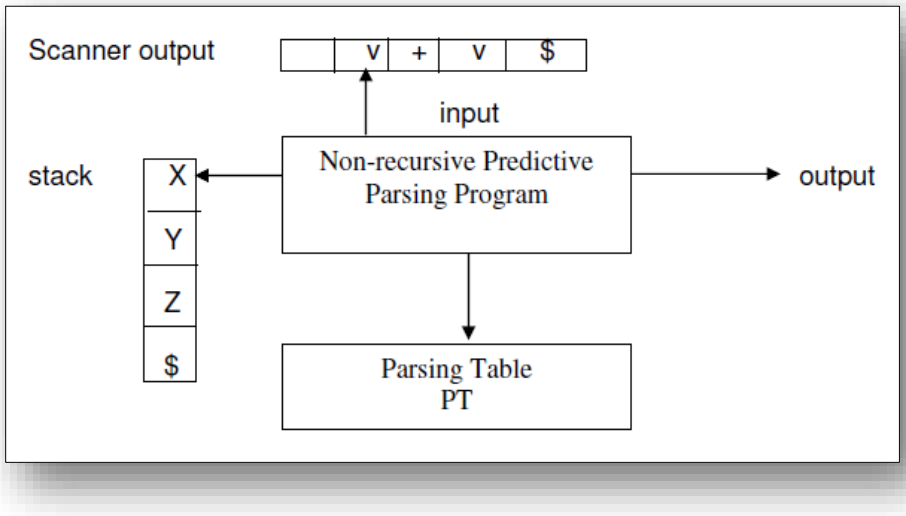


FIG 15.1: Scanning using stacks

The components are:

- input token stream terminated with $
- stack of grammar tokens with the $ at the bottom
- parsing table (called frequently transition table) showing next production
- the parsing table is two-dimensional array PT[A,a], where A is a nonterminal and a is a terminal defined in the language grammar.

Initially the stack contains the start symbol of the grammar on top of $.

15.1.1. Operations

The program considers X, the symbol at the top of the stack, and a, the current input symbol. The subsequent action can be one of:

- if X = a = $ the parsing is complete.
- if X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
- if X is a nonterminal the program consults the table M[A,a] to lookup the production with which to replace X on the stack or else give an error.
- If X → UVW is the selected production, then X on the stack is replaced by UVW with U at the top of the stack.

Therefore, we need a parsing table for the grammar (just as we needed a transition table for the regular expression).

- The table entries are determined by the **FIRST** and **FOLLOW** sets of the grammar.
- The **FIRST** set determines a production if the input token can be matched to the FIRST set.
- The **FOLLOW** set determines a production if the production derives and therefore will disappear but can match to a following production that contains the token.

15.1.2. Building Predictive Parsing Tables

The following algorithm can be used to construct a predictive **Parser Table** (**PT**) for a language L(G) defined by a grammar G.

- **BPPT1**. For each production A → α of the grammar G, do steps 2 and 3.
- **BPPT2**. For each terminal *a* in the FIRST(α), add to PT[A,a].
- **BPPT3**. If ε is in FIRST(α), and $ is the FOLLOW(A), add A → α to PT[A,b].

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and \$ is in $\text{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, \$]$ as well.

FIG 15.2: Predictive parsing

15.1.3. Example

Grammar:

Original	Transformed
E → E + T | T	E → TE'
	E' → +TE' |
T → T * F | F	T → FT'
	T' → *FT' |
F → v | (E)	F → v | (E)

FIRST and FOLLOW sets:

Nonterminal	FIRST	FOLLOW
E	{ v , (}	{) , $ }
E'	{ + , }	{) , $ }
T	{ v , (}	{ +,) , $ }
T'	{ * , }	{ + ,) , $ }
F	{ v , (}	{ * , + ,) , $ }

Parsing Table

Nonterminal	Input Symbol (token)					
	v	+	*	(	)	$
E	TE'			TE'		
E'		+TE'				
T	FT'			FT'		
T'			*FT'			
F	v			(E)		

15.1.4. Predictive Parsing Step-by-step

Here is what is happening when you start processing a sentence. Suppose, for example, (a+b)*c using the PP (Predictive Parsing):

Stack	Input	Table [A,a]
$E	(v+v)*v$	[E,(] => E->TE'
$E'T	(v+v)*v$	[T,(] => T->FT'
$E'T'F	(v+v)*v$	[F,(] => F->(E)
$E'T')E(	(v+v)*v$	Match and Remove (
$E'T')E	v+v)*v$	[E,v] => E->TE'
$E'T')E'T	v+v)*v$	[T,v] => T->FT'
$E'T')E'T'F	v+v)*v$	[F,v] => F->v
$E'T')E'T'v	v+v)*v$	Match and Remove v
$E'T')E'T'	+v)*v$	[T',+] => T'-> ε
$E'T')E'	+v)*v$	[E',+] => E'->+TE'
$E'T')E'T+	+v)*v$	Match and Remove +
$E'T')E'T	v)*v$	
$E'T')E'T'F	v)*v$	
$E'T')E'T'v	v)*v$	
$E'T')E'T'	)*v$	
$E'T')E'	)*v$	
$E'T')	)*v$	
$E'T'	*v$	
$E'T'F*	*v$	
$E'T'F	v$	
$E'T'v	v$	
$E'T'	$	
$E'	$	[E',$] => E'-> ε
$	$	Match and Stop

FIG 15.2: Step-by-step parsing

15.2. Parsing if Statement Grammar

15.2.1. Some Samples

Now, let us consider the case we deal with one specific grammar: conditional clauses using or not else. Look this grammar:

```
<statement> →
        if <condition> then <statement>
        | if <condition> then <statement> else <statement>
        | <other statements>
<condition> → boolean value
```

The following is an abstract presentation of the same grammar:

```
S → i C t S | i C t S e S | O
C → b
O → a
```

This grammar is not an LL grammar because it contains a “left-factor”.

Grammar

Original	Transformed
S → i C t S | i C t S e S | O	S → i C t S S’ | O
	S’ → eS | ε
C → b	C → b
O → a	O → a

FIRST and FOLLOW sets

Nonterminal	FIRST	FOLLOW
S	{ i , a }	{ e , $ }
S’	{ e , ε }	{ e , $ }
C	{ b }	{ t }
O	{ a }	{ e , $ }

Parsing Table

Nonterminal	Input Symbol (token)					
	a	b	e	i	t	$
S	O			iCtSS’		
S’			eS ε			ε
C		b				
O	a					

The grammar is **ambiguous** because the [S’, e] cell contains **two entries**.

- A grammar whose parsing table has no multiple entries in a cell is said to be LL(1) grammar.

Another way to prove that the if grammar is ambiguous.

```
S → i C t S | i C t S e S | O C → b
O → a
```

Given the sentence:

if b then if b then a else a

or

i b t i b t a e a

The following two left-most derivations can be built to parse the sentence:

```
S  → i C t S
   → i b t S
   → i b t i C t S e S
   → i b t i b t S e S
   → i b t i b t O e S
   → i b t i b t a e S
   → i b t i b t a e O
   → i b t i b t a e a
```

and

```
S  → i C t S e S
   → i b t S e S
   → i b t i C t S e S
   → i b t i b t S e S
   → i b t i b t O e S
   → i b t i b t a e S
   → i b t i b t a e O
   → i b t i b t a e a
```

The grammar can be reworked to remove the ambiguity:

```
S → M | U
M → if C then M else M | O
U → if C then S | if C then M else U
```

15.2.2. LL(1) Grammar Definition

A grammar is an LL(1) grammar if and only if whenever the grammar has a production $A \rightarrow \alpha \mid \beta$ with two distinctive options α and β the following conditions hold:

- For no terminal a do both α and β have a in their **FIRST** sets.
- At most one of α and β can derive the empty string.
- If $\beta \Rightarrow \varepsilon$, then α does not have in its **FIRST** set any terminal which is in **FOLLOW**(A).

Review Questions

1. What is the principle of predictive parser?
2. What is the problem caused by if-statement?

Article 16 Error Handling

16.1. Error Handling

In this section, some small concepts about Error Handling are covered.

Common errors:

- **Lexical errors** include misspellings of identifiers, keywords, or operators and missing quotes around text intended as a string.
- **Syntactic errors** include misplaced semicolons or extra or missing braces. Another example: the appearance of a case statement without an enclosing switch is a syntactic error.
- **Semantic errors** include type mismatches between operators and operands, e.g., the return of a value in a void method.
- **Logical errors** can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

16.1.1. Error handling process

Basically, we have these steps:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

16.1.2. How to handle errors

General Idea:

- The parser can detect a much wider range of errors than the scanner:
- Incorrectly structured statements (eg. If statements with no condition expression)
- Problems with arithmetic expressions (eg. unbalanced parentheses)
- These errors are largely **syntax errors**.

Any error handling routines should have the **following features**:

- Errors should be reported accurately.
- It should recover from errors quickly, so that later errors can be detected

- It should not slow down processing of correct programs significantly

16.1.3. Errors and Parsing

The parser can detect a much wider range of errors than the scanner, such as:

- Incorrectly structured statements (ex: If statements with no condition expression)
- Problems with arithmetic expressions (ex: unbalanced parentheses)

These errors are largely syntax errors

- Any error handling routines should have the following features:
    - Errors should be reported accurately
    - It should recover from errors quickly, so that later errors can be detected
    - It should not slow down processing of correct programs significantly

16.1.4. Error Correction Strategies

- **Panic Mode** - When an error is detected, tokens are discarded until some type of synchronizing token is found (such as a ; in C). This is probably easiest to implement.
- **Phrase-Level Recovery** - Replace tokens with ones that would make the program syntactically correct and allow the parser to continue (eg. substitute a ; for a comma if it is where the end of a statement should be)
    - Build error productions into the grammar (assuming common errors are known before hand).
- **Global corrections** - Looks at input, finds the least number of changes necessary to produce a valid string.

16.2. Error Handling

16.2.1. Parsing Table with Error Recovery Synchronization Entries

Process

The following steps are describing the process for Error Recovery:

- As a starting point, place all symbols in **FOLLOW**(A) into the synchronizing set for nonterminal A.
  - If we skip tokens until an element of **FOLLOW**(A) is seen and pop A from the stack, it is likely that parsing can continue.
- It is not enough to use **FOLLOW**(A) as the synchronizing set for A.
  - For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the **FOLLOW** set of the nonterminal expressions.
  - A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.
  - Often, there is a hierarchical structure on constructs in a language; for example, expressions appear within statements, which appear within blocks, and so on.
  - We can add to the synchronizing set of a lower-level construct the symbols that begin higher-level constructs.
  - For example, we might add keywords that begin statements to the synchronizing sets for the non-terminals generating expressions.
- If we add symbols in **FIRST**(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in **FIRST**(A) appears in the input.
- If a nonterminal can generate the empty string, then the production deriving e can be used as a default.
  - Doing so may postpone some error detection, but cannot cause an error to be missed.
  - This approach reduces the number of nonterminal that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

PT Example

Now that we have the Predictive Parsing Table, it is also possible to identify and treat errors using recovery by synchronization:

Nonterminal	Input Symbol (token)					
	v	+	*	(	)	$
E	TE’			TE’	synch	synch
E’		+TE’				
T	FT’	synch		FT’	synch	synch
T’			*FT’			
F	v	synch	synch	(E)	synch	synch

Review Questions

1. What is the principle of error handling?
2. How are the advantages of using different approaches for error detection?
3. How panic mode is basically implemented?

Part II

Annexes

Annex 1 — Useful Tables

Here are some tables that can help you…

Table 1 – ASCII Chars table

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

FIG AN2.1: ASCII Table - Source: https://simple.wikipedia.org/wiki/ASCII

Table 2 – Initial Unicode table

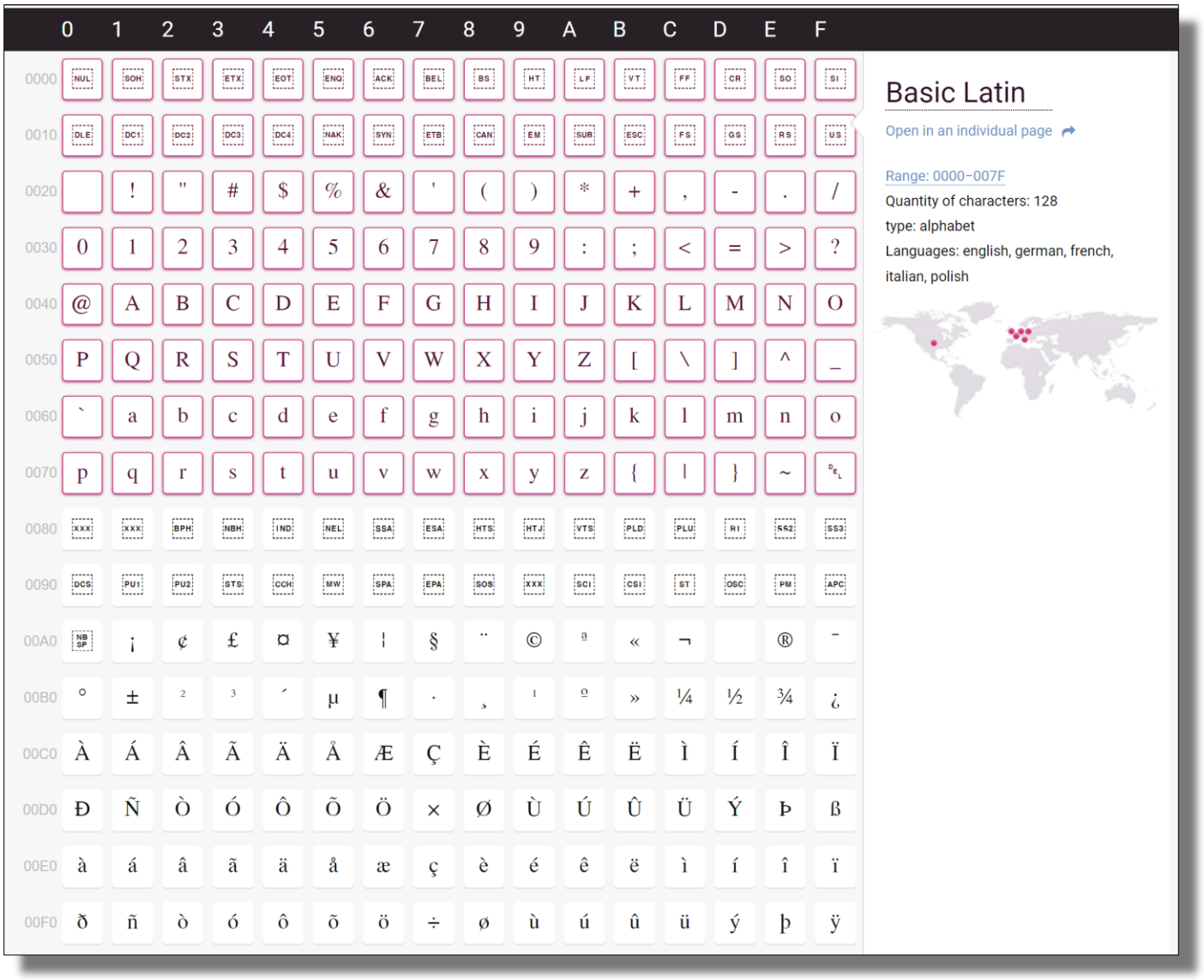
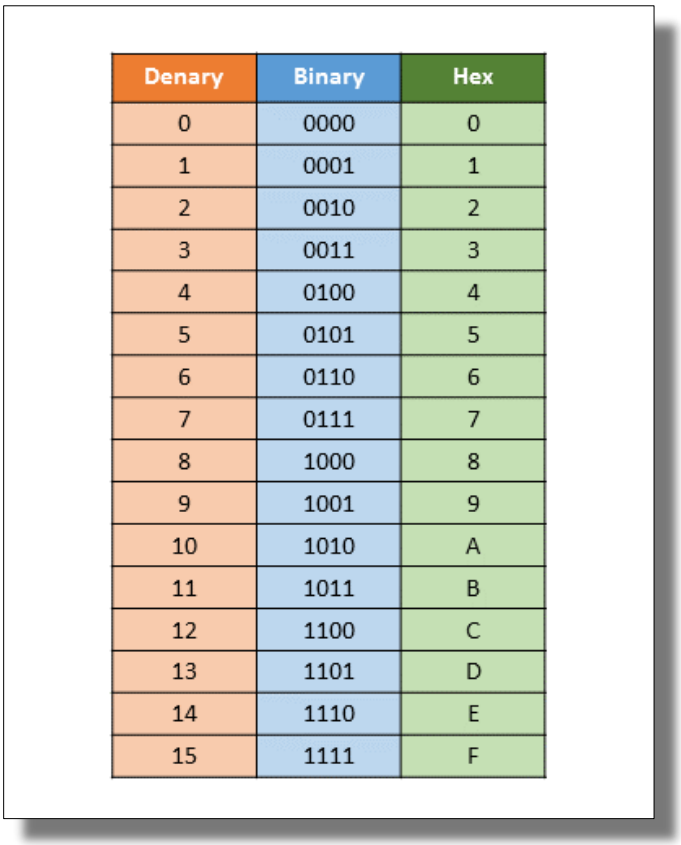


FIG AN2.2: UNICODE table - Source: https://unicode-table.com/en/

Table 3 – Hex / Bin / Dec Table



Denary	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

FIG AN2.3: HEXA code - Source: http://wiki.schoolcoders.com/gcse/data-representation/numbers/hexadecimal/

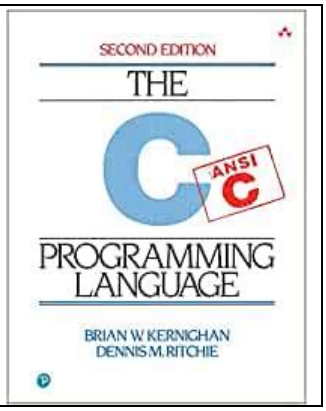
Annex 2 — Useful C Language Concepts

1.1. Preprocessor - Basic Notation

What is pre-processing?

C provides certain **language facilities** by means of a preprocessor, which is conceptionally a separate first step in compilation.

- The two most frequently used features are #include, to include the contents of a file during compilation, and #define, to replace a token by an arbitrary sequence of characters.
- Other features include conditional compilation and macros with arguments.



1. File inclusion

```
#include <filename>        /* standard libraries header files */
#include "filename"    /* user defined header files */
```

2. Macro Definitions and Expansions (Macro Substitutions)

```
#define NAME                          /* defines name */
#undef NAME                           /* undefines name */
#define NAME  Replacement             /* replaces NAME with replacement */
#define name(list) expression     /* inline functions or macro */
```

3. Conditional processing (compilation)

```
#if constant-expression

#ifdef NAME

#if defined(NAME)

#ifndef NAME
#if !defined(NAME)
#elif constant-expression /* optional */
#else                             /* optional */
#endif                            /* must be present */
```

4. Pragmas (Pragmatics)

```
#pragma directive      /* sets different  compiler options */
```

5. Other

```
#                     /* null directive – no any action or "" enclosure */
##                    /* token passing – concatenation */
#line constant       /* changes the line number of the source */
#error text          /* compiler displays error message including text */
```

6. Predefined macros in ANSI C:

```
__LINE_____,___FILE____,__TIME_____,___DATE_____,___STDC_____
```

1.2. Preprocess Examples

9. File inclusion

```
#include <limits.h>
```

The header file must be in the *include* directory of the compiler

```
#include "buffer.h"
```

The header file must be in the same directory as the source file containing the `include` preprocessor directive

10. Protecting header files from multiple inclusions

```
#ifndef BUFFER_H_
#define BUFFER_H_
/* ... declaration */
#endif
```

When the header file is included for the first time BUFFER_H_ is not defined and the contents of the file (the declarations) is included in the source file which contains the include statement. If the header file is included for the second time (this happens when you work with multiple source file programs), BUFFER_H_ has already been defined and the declarations will not be included. This prevents from duplicate declarations syntax errors.

11. Inserting debug statements in a program

The following line must be placed in your header file or on the top of your source file

```
#define DEBUG
```

If you want to print some debugging information in your program, you include the following lines

```
#ifdef DEBUG
          printf("The size is negative: %d", size)
#endif
```

When the testing of your program is finished you must the define statement from the header or place somewhere under the define

```
#undef DEBUG
```

The #undef directive is often used to suppress macros when a routine or procedure is implemented both as a macro and as a function.

```
#undef getchar
int getchar(void){…}
```

12. Defining constants in C

```
#define FAIL -1
```

The word `FAIL` will be replaced with `-1` everywhere in the program starting from the line following the define directive to the end of the source file.

- **Warning**: If `FAIL` is enclosed in double quotes it will not be replaced. If, however, a parameter name is preceded by a # symbol in the replacement text, the combination will be replaced by the actual argument enclosed in quotes.

```
#define reprint(fexp) printf(#fexp " = %9.3f\n",fexp)
```

When the macro (see below) is invoked, as in

```
reprint(x+y);
```

it will be expanded to

```
printf("x + y" " = %9.3f\n",x+y);
```

13. Defining macros

It is possible to define macros, which resembles functions. These pseudo functions are truly generic because data of any type can be used with them as long as the macro contains the proper statement. If the #define value can not fit on one line and must be carried over another line, then the backslash (\) character should be the last thing on the line. For example:

```
#define WCALC(y,x) \
((y + x) / (x -y) * \
(INT_MAX - 77x))
```

Once defined, you can use them as normal functions.

```
#define sum(x,y) x+y
 a = a + sum(b,c);
```

The preprocessor will replace the function-like call with the replacement text. The `sum()` in the line above will be replaced with b+c:

```
a = a + b+c;
```

This looks like inline functions in C++, but it is not the same. When writing macros in C, you must remember that the preprocessor simply replaces the macro. This could lead to serious mistakes. For example, if you write the following

```
a = a * sum(b+c,d+e) / m;
```

The result from the substitution is

```
a = a * b+c + d+e / m;
```

As you see, this is wrong. To avoid these kind of mistakes always use parentheses

```
#define sum(x,y)          ((x) + (y))
```

Now the result is correct

```
a = a * ((x) + (y)) / m;
```

The directive ## can be used to create variable names. If the following is used:

```
#define STR(a,b)          a ## b
...
STR(str,1) = strcat(STR(str,2), STR(my,name));
```

The result after the preprocessor will be

```
str1 = strcat(str2, myname);
```

14. Including a proper file (header)

The following sequence will check the type of the application and will include the proper header file.

```
#define PLATFORM 10
...
#if    PLATFORM ==    __MSDOS__ *__10
#define HEADER         "dos.h"
#elif PLATFORM == __WIN32__* 9
```

```
#define HEADER          "win.h"
#else
#define HEADER          "sys.h"
#endif
#include HEADER
```

- **Note:** __MSDOS__ and __WIN32__ are Borland specific (non ANSI) manifest constants.

15. Enforcing the same data type size across different platforms

```
#define BORLAND_16
#ifdef BORLAND_16
   typedef int int_2b;
   typedef long long_4b;
#endif
#ifdef MSVS_16
   typedef short int_2b;
   typedef int long_4b;
#endif

printf("Int size: %d Long size: %d\n",
        sizeof(int_2b), sizeof(long_4b));
```

16. Pragmas (Pragmatics)

```
#pragma inline
```

This line will force the compiler (Borland C/C++) to generate assembler output and call the assembler.

```
#pragma warning(disable : 4996)
```

This line will suppress the warning by the Visual Studio C/C++ compiler which is generated when a deprecated function is used in the code.

- For example, if the `string.h` declared functions are used in a C/C++ code compiled by MS Visual Studio, C4996 warning will be generated.
- C4996 is generated for the line on which the function is declared and for the line on which the function is used.
- If a compiler does not recognize the `pragma` directive, it simply ignores it. This convention allows different compilers to use different `pragma` directives.

If you want to learn more about the C Language, you can make a trip to:

http://www.lysator.liu.se/c/

You can learn there about different C Language Standards and much more.

1.3. Bitwise operations

A **bitwise operation** operates on one or more-bit patterns or binary numerals at the level of their individual bits. It is a fast and simple action, directly supported by the processor, and is used to manipulate values for comparisons and calculations.

- **Note:** Bitwise operators treat their operands as a sequence of bits (zeroes and ones), rather than as decimal numbers. They operate on a single bit in the same location of the operands.

```
char c = -1;
```

1	1	1	1	1	1	1	1	1

```
c = 1;
```

0	0	0	0	0	0	0	0	1

Truth table – Boolean operations

X1	X2	&	I	^	~X1	~X2
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

EXAMPLES:

Setting flag to 0

Flag variable

1	1	1	1	1	1	1	1	1

Mask:

0	0	0	0	0	0	0	0	0

AND operation = Result

0	0	0	0	0	0	0	0	0

Setting flag to 1

Flag variable

1	1	1	1	1	1	1	1	1

Mask:

0	0	0	0	0	0	0	0	0

AND operation = Result

0	0	0	0	0	0	0	0	0

Mask:

0	0	0	0	0	0	0	0	1

OR operation = Result

0	0	0	0	0	0	0	0	1

Checking flag to 1

Flag variable

1	1	1	1	1	1	1	1	1

Mask:

0	0	0	0	0	0	0	0	1

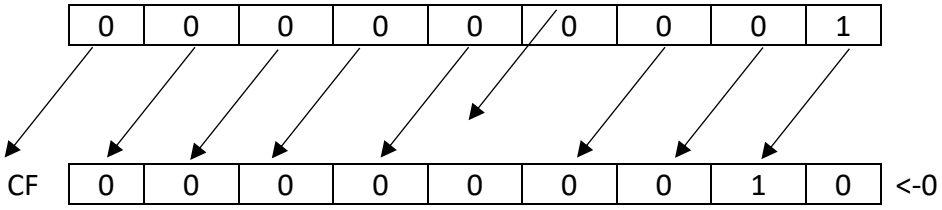
AND operation = Result

0	0	0	0	0	0	0	0	1

Other bitwise operations – left and right logical and arithmetic shifts

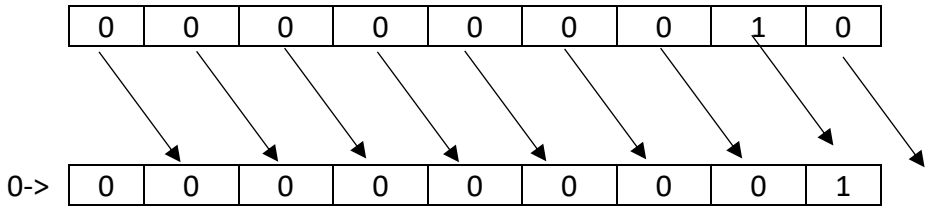
<< left shift - logical and arithmetic

```
char c = 1;
c << 1;
```



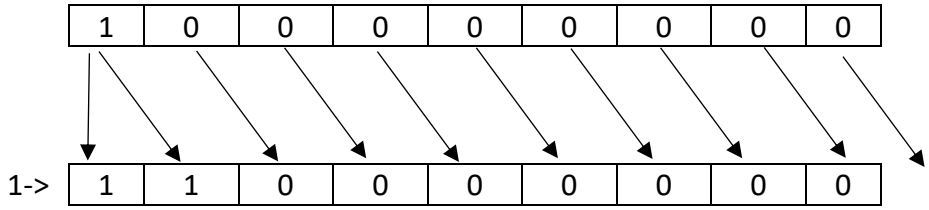
>> right shift – arithmetic

```
char   c = 2;
c >> 1;
```



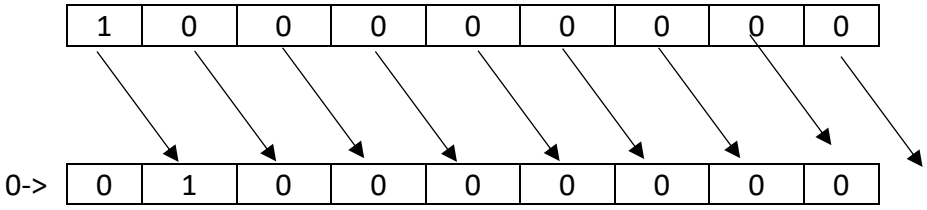
>> right shift – arithmetic

```
char   c = 128;
c >> 1;
```



>> right shift – logical

```
unsigned char c = 128;
c >> 1;
```



Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include −

- **NOTE:** Packing several objects into a machine word. e.g. 1 bit flags can be compacted.

C allows us to do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:2;
} pack;
```

Here, the `packed_struct` contains 6 members: Four 1 bit flags f1..f3, a 4-bit type. C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

1.4. User Defined Data types

User Defined Data types – structure data type:

2. **Declaration – syntax**

Example:

```
struct s {int a; char c};
struct {int a; char c} s;
```

3. **Definition**

Example:

```
struct S s;
```

- **NOTE:** To avoid using struct keyword all the time, use typedef

Example:

```
typedef struct Structure {int a; char c} S;
S s;
```

4. **Operations**

Example:

```
. , →, *, &, =
```

5. **Initialization**

Example:

```
struct S s = {5,'S'};
```

5.1. C-Language Variable Attributes

A C program consists of a set of external objects, which are either variables or functions. Those objects can be grouped in different source files or compilation units thus creating a multiple-source file program.

- Variables must be defined outside any function in contrast to the function arguments and variables defined inside functions, that is, they must be "external" or "global" to any function in contrast to the "internal" variables.
- Functions in C are always external, because C does not allow functions to be defined inside other functions.
- A multiple source file program need not all be compiled at one time and precompiled functions may be loaded from libraries.
- Communications among the functions of a program may be carried out both trough function calls and through manipulation of external variables.

C variables have the following attributes: **identifier** (name), **type**, **size**, **mutability**, storage class (**lifetime**), scope (**visibility**), and **linkage**. The type or the value of an attribute is specified in a C program through different keywords – modifiers and specifiers.

The following table summarizes the C-language variable storage class, scope and linkage attributes.

Specifier/ Modifier	Storage class Lifetime	Linkage	Point of Declaration/ Definition	Scope (visibility)	How many variables with the same name in
auto register volatile (or none)	automatic (local) -- block lifetime function lifetime	internal	inside a block or function (local variable or parameter)	block or function	one for each block or function. -- for nested blocks the inner variable
(none) **volatile extern**	**static** (global) -- program lifetime	external	outside a function -- in other source files or functions must be declared as **extern** -- functions are **extern** by default	from the point of definition / declaration to the end of the file. -- Program must be declared **extern** in another source file	one in a file many if declared **extern** in a function or in other files
static volatile	**static** -- program lifetime	internal	inside a block or function	block or function -- retains its value when out of scope	one in a block or function
static volatile	**static** (global) -- program lifetime	internal	outside a function -- can be applied to a function	from the point of definition to the end of the file.	one in a file one in a program can be used only in one file

- A variable **mutability** determines whether and how the initial value of a variable can be changed in a program. Keywords: const, register, volatile, static.
- An identifier **storage class** (lifetime) determines the period during which that identifier exists in memory. There are two storage classes in C – automatic and static. Automatic variables have a function or a block lifetime (local lifetime). Static variables have a program lifetime. Keywords: auto, register, extern, static,
- An identifier scope (**visibility**) determines where the identifier can be referenced in a program or where the name can be used. There are four different kinds of scope (lexical scope): file scope, function scope (labels only), block scope, and function declaration scope (argument list only).
    - In multiple-source file program there is another scope associated with variables and functions with external linkage, which determines the connection between identifiers in separately compiled units.

- Identifier falls into several different name spaces that do not interfere with one another, that is, the same identifier may be used for different purposes even in the same scope.
- **Example**: variable names, function names, typedef names, and enum constants; tags of structures, unions, and enumerations; members of each structure or union individually. Keywords: static, extern.

- An identifier linkage determines for a multiple-source file program whether an identifier is known only in the current source file or in any source file with proper declarations.
  - There are two kinds of linkage: external and internal. Keywords: static, extern.

Final message:

- C is not so hard:
  - `void ( * ( * f [ ] ) ( ) ) ( )` defines f as an array of unspecified size, of pointers to functions that return pointers to functions that return void.

Annex 3 — Complete ANSI C Reference Card

C Reference Card (ANSI)

Program Structure/Functions

function declarations	***type fnc*(*type1*, …)**
external variable declarations	***type name***

main routine
  local variable declarations
	main() {
	  declarations
	  statements
	}

function definition
  local variable declarations
	***type fnc*(*arg1*, …) {**
	  declarations
	  statements
	  return *value*;
	}

comments	**/* */**
main with args	**main(int argc, char *argv[])**
terminate execution	**exit(*arg*)**

C Preprocessor

include library file	**#include <*filename*>**
include user file	**#include "*filename*"**
replacement text	**#define *name text***
replacement macro	**#define *name*(*var*) *text***
Example: *#define max(A,B) ((A)>(B) ? (A) : (B))*	
undefine	**#undef *name***
quoted string in replace	**#**
concatenate args and rescan	**##**
conditional execution	**#if, #else, #elif, #endif**
is name defined, not defined?	**#ifdef, #ifndef**
name defined?	**defined(*name*)**
line continuation char	****

Data Types/Declarations

character (1 byte)	**char**
integer	**int**
float (single precision)	**float**
float (double precision)	**double**
short (16 bit integer)	**short**
long (32 bit integer)	**long**
positive and negative	**signed**
only positive	**unsigned**
pointer to int, float, …	***int, *float, …**
enumeration constant	**enum**
constant (unchanging) value	**const**
declare external variable	**extern**
register variable	**register**
local to source file	**static**
no value	**void**
structure	**struct**
create name by data type	**typedef *typename***
size of an object (type is size_t)	**sizeof *object***
size of a data type (type is size_t)	**sizeof(*type name*)**

Initialization

initialize variable type	***name*=*value***
initialize array	***type name*[]={*value1* ,… }**
initialize char string	**char *name*[]="*string*"**

Constants

long (suffix)	**L** or **l**
float (suffix)	**F** or **f**
exponential form	**e**
octal (preffix zero)	**0**
hexadecimal (preffix zero-ex)	**0x** or **0X**
character constant (char, octal, hex)	**'a'**, **'\o*oo*'**, **'\x*hh*'**
newline, cr, tab, backspace	**\n**, **\r**, **\t**, **\b**
special characters	****, **\?**, **\'**, **\"**
string constant (ends with **'\0'**)	**"abc…de"**

Pointers, Arrays & Structures

declare pointer to type	***type* *name***
declare function returning pointer to type	***type* *f()***
declare pointer to function returning type	***type* (*pf) ()***
generic pointer type	**void ***
null pointer	**NULL**
object pointed to by pointer	***pointer***
address of object name	**&*name***
array	***name*[*dim*]**
multi-dim array	***name*[*dim1*][*dim2*]…**

Structures

structure template
  declaration of members
	struct *tag* {
	  declarations
	};

create structure	**struct *tag name***
member of structure from template	***name*.*member***
member of pointed to structure	***pointer* -> *member***
***Example*: (*p).x** and **p->x** are the same	
single value, multiple type structure	**union**
bit field with b bits	***member* : *b***

Operators (grouped by precedence)

structure member operator	***name*.*member***
structure pointer	***pointer*->*member***
increment, decrement	**++, --**
plus, minus, logical not, bitwise not	**+, -, !, ~**
indirection via pointer, address of object	***pointer*, &*name***
cast expression to type	**(*type*) *expr***
size of an object	**sizeof**
multiply, divide, modulus (remainder)	***, /, %**
add, subtract	**+, -**
left, right shift [bit ops]	**<<, >>**
comparisons	**>, >=, <, <=**
comparisons	**==, !=**
bitwise and	**&**
bitwise exclusive or	**^**
bitwise or (incl)	**|**
logical and	**&&**
logical or	**||**
conditional expression	***expr1* ? *expr2* : *expr3***
assignment operators	**+=, -=, *=, …**
expression evaluation separator	**,**

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	**;**
block delimeters	**{ }**
exit from switch, while, do, for	**break**
next iteration of while, do, for	**continue**
go to	**goto *label***
label	***label*:**
return value from function	**return *expr***

Flow Constructions

if statement

```
if (expr) statement
else if (expr) statement
else statement
```

while statement

```
while (expr)
  statement
```

for statement

```
for (expr1; expr2; expr3)
  statement
```

do statement

```
do
  statement
while(expr);
```

switch statement

```
switch (expr) {
  case const1:
    statement1
    break;
  case const2:
    statement2
    break;
  default:
    statement
}
```

ANSI Standard Libraries

<assert.h> <ctype.h> <errno.h> <float.h> <limits.h>
<locale.h> <math.h> <setjmp.h> <signal.h> <stdarg.h>
<stddef.h> <stdio.h> <stdlib.h> <string.h> <time.h>

Character Class Tests <ctype.h>

alphanumeric?	**isalnum(c)**
alphabetic?	**isalpha(c)**
control character?	**iscntrl(c)**
decimal digit?	**isdigit(c)**
printing character (not incl space)?	**isgraph(c)**
lower case letter?	**islower(c)**
printing character (incl space)?	**isprint(c)**
printing char except space, letter, digit?	**ispunct(c)**
space, formfeed, newline, cr, tab, vtab?	**isspace(c)**
upper case letter?	**isupper(c)**
hexadecimal digit?	**isxdigit(c)**
convert to lower case?	**tolower(c)**
convert to upper case?	**toupper(c)**

String Operations <string.h>

***NOTE**: s,t are strings, cs, ct are constant strings*

length of s	**strlen(s)**
copy ct to s	**strcpy(s,ct)**
up to n chars	**strncpy(s,ct,n)**
concatenate ct after s	**strcat(s,ct)**
up to n chars	**strncat(s,ct,n)**
compare cs to ct	**strcmp(cs,ct)**
only first n chars	**strncmp(cs,ct,n)**
pointer to first c in cs	**strchr(cs,c)**
pointer to last c in cs	**strrchr(cs,c)**
copy n chars from ct to s	**memcpy(s,ct,n)**
copy n chars from ct to s (may overlap)	**memmove(s,ct,n)**
compare n chars of cs with ct	**memcmp(cs,ct,n)**
pointer to first c in first n chars of cs	**memchr(cs,c,n)**
put c into first n chars of cs	**memset(s,c,n)**

Input/Output <stdio.h>

Standard I/O

standard input stream	**stdin**
standard output stream	**stdout**
standard error stream	**stderr**
end of file	**EOF**
get a character	**getchar()**
print a character	**putchar(*chr*)**
print formatted data	**printf("*format* ", *arg1*, …)**
print to string s	**sprintf(s,"*format* ", *arg1*, …)**
read formatted data	**scanf("*format*", &*name1*, …)**
read from string s	**sscanf(s,"*format*",&*name1*, …)**
read line to string s (< max chars)	**gets(s,max)**
print string s	**puts(s)**

File I/O

declare file pointer	**FILE *fp**
pointer to named file	**fopen("*name*","*mode*")**
modes: **r** (read), **w** (write), **a** (append)	
get a character	**getc(fp)**
write a character	**putc(chr ,fp)**
write to file	**fprintf(fp,"*format*", *arg1*, …)**
read from file	**fscanf(fp,"format", *arg1*, …)**
close file	**fclose(fp)**
non-zero if error	**ferror(fp)**
non-zero if EOF	**feof(fp)**
read line to string s (< max chars)	**fgets(s,max,fp)**
write string s	**fputs(s,fp)**

Codes for Formatted I/O: "%-+ 0w:pmc"

left justify	**-**
print with sign	**+**
print space if no sign	***space***
pad with leading zeros	**0**
min field width	**w**
precision	**p**
conversion character:	**m**
h short, **l** long,	**L** long double
conversion character:	**c**
d,i integer,	**u** unsigned
c single char,	**s** char string
f double,	**e,E** exponential
o octal	**x,X** hexadecimal
p pointer	**n** number of chars written
g,G same as f or	**e,E** depending on exponent

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	**va_list *name*;**
initialization of argument pointer	**va_start(*name*, *lastarg*)**
Note:* *lastarg *is last named parameter of the function*	
access next unamed arg, update pointer	**va_arg(*name*, *type*)**
call before exiting function	**va_end(*name*)**

Standard Utility Functions <stdlib.h>

absolute value of int n	**abs(n)**
absolute value of long n	**labs(n)**
quotient and remainder of ints n,d	**div(n,d)**
returns structure with div_t.quot and div_t.rem	
quotient and remainder of longs n,d	**ldiv(n,d)**
returns structure with ldiv_t.quot and ldiv_t.rem	
pseudo-random integer [0, RAND_MAX]	**rand()**
set random seed to n	**srand(n)**
terminate program execution	**exit(*status*)**
pass string s to system for execution	**system(s)**

Conversions

convert string s to double	**atof(s)**
convert string s to integer	**atoi(s)**
convert string s to long	**atol(s)**
convert prefix of s to double	**strtod(s,endp)**
convert prefix of s (base b) to long	**strtol(s,endp,b)**
same, but unsigned long	**strtoul(s,endp,b)**

Storage Allocation

allocate storage	**malloc(size), calloc(nobj,size)**
change size of object	**realloc(pts,size)**
deallocate space	**free(ptr)**

Array Functions

search array for key	**bsearch(key,array,n,size,cmp())**
sort array ascending order	**qsort(array,n,size,cmp())**

Time and Date Functions <time.h>

processor time used by program	**clock()**
Example*. clock()/CLOCKS_PER_SEC is time in seconds*	
current calendar time	**time()**
Note: *time2-time1 in seconds (double) difftime(time2,time1)*	
arithmetic types representing times	**clock_t, time_t**
structure type for calendar time comps	**tm**
tm_sec seconds after minute	
tm_min minutes after hour	
tm_hour hours since midnight	
tm_mday day of month	
tm_mon months since January	
tm_year years since 1900	
tm_wday days since Sunday	
tm_yday days since January 1	
tm_isdst Daylight Savings Time ag	
convert local time to calendar time	**mktime(tp)**
convert time in tp to string	**asctime(tp)**
convert calendar time in tp to local time	**ctime(tp)**
convert calendar time to GMT	**gmtime(tp)**
convert calendar time to local time	**localtime(tp)**
format date and time info	**strftime(s,smax,"*format* ",tp)**
Note: *tp is a pointer to a structure of type tm*	

Mathematical Functions <math.h>

Note: *Arguments and returned values are double*

trig functions	**sin(x), cos(x), tan(x)**
inverse trig functions	**asin(x), acos(x), atan(x)**
arctan(y=x)	**atan2(y,x)**
hyperbolic trig functions	**sinh(x), cosh(x), tanh(x)**
exponentials & logs	**exp(x), log(x), log10(x)**
exponentials & logs (2 power)	**ldexp(x,n), frexp(x,*e)**
division & remainder	**modf(x,*ip), fmod(x,y)**
powers	**pow(x,y), sqrt(x)**
rounding	**ceil(x), floor(x), fabs(x)**

Integer Type Limits <limits.h>

Note: *The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.*

CHAR_BIT bits in **char**	(8)
CHAR_MAX max value of char	(127 or 255)
CHAR_MIN min value of char	(-128 or 0)
INT_MAX max value of int	(+32,767)
INT_MIN min value of int	(-32,768)
LONG_MAX max value of long	(+2,147,483,647)
LONG_MIN min value of long	(-2,147,483,648)
SCHAR_MAX max value of signed char	(+127)
SCHAR_MIN min value of signed char	(-128)
SHRT_MAX max value of short	(+32,767)

SHRT_MIN min value of short	(-32,768)
UCHAR_MAX max value of unsigned char	(255)
UINT_MAX max value of unsigned int	(65,535)
ULONG_MAX max value of unsigned long	(4,294,967,295)
USHRT_MAX max value of unsigned short	(65,536)

Standard Limits <stdlib.h>

Note: *Some definitions are implementation dependent.*

RAND_MAX max value for random	(32767)

Float Type Limits <float.h>

FLT_RADIX radix of exponent rep	(2)
FLT_ROUNDS floating point rounding mode	
FLT_DIG decimal digits of precision	(6)
FLT_EPSILON smallest x so 1:0 + x 6= 1:0	(10**-5)
FLT_MANT_DIG number of digits in mantissa	
FLT_MAX maximum floating point number	(10**+37)
FLT_MAX_EXP maximum exponent	
FLT_MIN minimum floating point number	(10**-37)
FLT_MIN_EXP minimum exponent	

DBL_DIG decimal digits of precision	(10)
DBL_EPSILON smallest x so 1:0 + x 6= 1:0	(10**-9)
DBL_MANT_DIG number of digits in mantissa	
DBL_MAX max double floating point num	(10**+37)
DBL_MAX_EXP maximum exponent	
DBL_MIN min double floating point num	(10**-37)
DBL_MIN_EXP minimum exponent	

Annex 4 — Useful Links

Here are some useful links that students can consider for additional information about compilers…

Chapter 1

- **About Programming Languages:**
  - TIOBE site:
    - https://www.tiobe.com/tiobe-index/
  - Hierarchy / History of Programming Languages:
    - http://blog.daveastels.com.s3-website-us-west-2.amazonaws.com/languages.html

- **To think about the future:**
  - Facebook AI creates its own language in creepy preview:
    - https://www.forbes.com/sites/tonybradley/2017/07/31/facebook-ai-creates-its-own-language-in-creepy-preview-of-our-potential-future/
  - The truth behind Facebook AI inventing a new language:
    - https://towardsdatascience.com/the-truth-behind-facebook-ai-inventing-a-new-language-37c5d680e5a7
  - OpenAI API:
    - https://openai.com/blog/openai-api/
  - GPT-3 Demo:
    - https://www.youtube.com/watch?v=8psgEDhT1MM
  - GPT-3 Paper:
    - https://arxiv.org/pdf/2005.14165.pdf
  - Kevin Lacker tests:
    - https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html
  - ChatGPT:
    - https://openai.com/blog/chatgpt/
    - https://arxiv.org/abs/2203.02155

Chapter 3

- **About datatypes:**
  - The problem about wrong datatype manipulation (contribution of Martin Weaver):
    - https://screenrant.com/civilization-gandhi-evil-civ-6-glitch-nukes-why/

Appendixes

- **ASCII Table:**
  - Link: https://simple.wikipedia.org/wiki/ASCII

- **Unicode Table:**
  - Link: https://unicode-table.com/en/

Algonquin College
Fall, 2023

Part III

Language Specification

Lang 1

Go Language Specification

Introduction

Go is a general-purpose language designed with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming. Programs are constructed from *packages*, whose properties allow efficient management of dependencies.

The syntax is compact and simple to parse, allowing for easy analysis by automatic tools such as integrated development environments.

Notation

The syntax is specified using a variant of Extended Backus-Naur Form (EBNF):

```
Syntax      = { Production } .
Production  = production_name "=" [ Expression ] "." .
Expression  = Term { "|" Term } .
Term        = Factor { Factor } .
Factor      = production_name | token [ "…" token ] | Group |
Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

Lowercase production names are used to identify lexical (terminal) tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes `""` or back quotes ``` ``.

The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis `…` is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character `…` (as opposed to the three characters ...) is not a token of the Go language.

Characters

The following terms are used to denote specific Unicode character categories:

```
newline        = /* the Unicode code point U+000A */ .
unicode_char   = /* an arbitrary Unicode code point except newline
*/ .
unicode_letter = /* a Unicode code point categorized as "Letter" */
.
unicode_digit  = /* a Unicode code point categorized as "Number,
decimal digit" */ .
```

In The Unicode Standard 8.0, Section 4.5 "General Category" defines a set of character categories. Go treats all characters in any of the Letter categories Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in the Number category Nd as Unicode digits.

Letters and digits

The underscore character _ (U+005F) is considered a lowercase letter.

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
binary_digit  = "0" | "1" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

Lexical elements

Comments

Comments serve as program documentation. There are two forms:

1. *Line comments* start with the character sequence `//` and stop at the end of the line.
2. *General comments* start with the character sequence `/*` and stop with the first subsequent character sequence `*/`.

A comment cannot start inside a rune or string literal, or inside a comment. A general comment containing no newlines acts like a space. Any other comment acts like a newline.

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*. *White space*, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal syntax uses semicolons `";"` as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is
    - an identifier
    - an integer, floating-point, imaginary, rune, or string literal
    - one of the keywords `break`, `continue`, `fallthrough`, or `return`
    - one of the operators and punctuation `++`, `--`, `)`, `]`, or `}`
2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing `")"` or `"}"`.

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
a
_x9
ThisVariableIsExported
αβ
```

Some identifiers are predeclared.

Keywords

The following keywords are reserved and may not be used as identifiers.

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=          ~
```

Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0b` or `0B` for binary, `0`, `0o`, or `0O` for octal, and `0x` or `0X` for hexadecimal. A single `0` is considered a decimal zero. In hexadecimal literals, letters `a` through `f` and `A` through `F` represent values 10 through 15.

```
int_lit        = decimal_lit | binary_lit | octal_lit | hex_lit .
decimal_lit    = "0" | ( "1" … "9" ) [ [ "_" ] decimal_digits ] .
binary_lit     = "0" ( "b" | "B" ) [ "_" ] binary_digits .
octal_lit      = "0" [ "o" | "O" ] [ "_" ] octal_digits .
hex_lit        = "0" ( "x" | "X" ) [ "_" ] hex_digits .

decimal_digits = decimal_digit { [ "_" ] decimal_digit } .
binary_digits  = binary_digit { [ "_" ] binary_digit } .
octal_digits   = octal_digit { [ "_" ] octal_digit } .
hex_digits     = hex_digit { [ "_" ] hex_digit } .
42
4_2
0600
0_600
0o600
```

```
0O600       // second character is capital letter 'O'
0xBadFace
0xBad_Face
0x_67_7a_2f_cc_40_c6
170141183460469231731687303715884105727
170_141183_460469_231731_687303_715884_105727

_42         // an identifier, not an integer literal
42_         // invalid: _ must separate successive digits
4__2        // invalid: only one _ at a time
0_xBadFace  // invalid: _ must separate successive digits
```

Floating-point literals

A floating-point literal is a decimal or hexadecimal representation of a floating-point constant.

A decimal floating-point literal consists of an integer part (decimal digits), a decimal point, a fractional part (decimal digits), and an exponent part (`e` or `E` followed by an optional sign and decimal digits). One of the integer part or the fractional part may be elided; one of the decimal point or the exponent part may be elided. An exponent value exp scales the mantissa (integer and fractional part) by 10^{exp}.

```
float_lit         = decimal_float_lit | hex_float_lit .

decimal_float_lit = decimal_digits "." [ decimal_digits ] [
decimal_exponent ] |
                    decimal_digits decimal_exponent |
                    "." decimal_digits [ decimal_exponent ] .
decimal_exponent  = ( "e" | "E" ) [ "+" | "-" ] decimal_digits .

hex_float_lit     = "0" ( "x" | "X" ) hex_mantissa hex_exponent .
hex_mantissa      = [ "_" ] hex_digits "." [ hex_digits ] |
                    [ "_" ] hex_digits |
                    "." hex_digits .
hex_exponent      = ( "p" | "P" ) [ "+" | "-" ] decimal_digits .
0.
72.40
072.40       // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
1_5.         // == 15.0
0.15e+0_2    // == 15.0

0x1p-2       // == 0.25
0x2.p10      // == 2048.0
0x1.Fp+0     // == 1.9375
0X.8p-0      // == 0.5
0X_1FFFP-16  // == 0.1249847412109375
0x15e-2      // == 0x15e - 2 (integer subtraction)

0x.p1        // invalid: mantissa has no digits
1p-2         // invalid: p exponent requires hexadecimal mantissa
0x1.5e-2     // invalid: hexadecimal mantissa requires p exponent
1_.5         // invalid: _ must separate successive digits
1._5         // invalid: _ must separate successive digits
1.5_e1       // invalid: _ must separate successive digits
1.5e_1       // invalid: _ must separate successive digits
1.5e1_       // invalid: _ must separate successive digits
```

Imaginary literals

An imaginary literal represents the imaginary part of a complex constant. It consists of an integer or floating-point literal followed by the lowercase letter `i`. The value of an imaginary literal is the value of the respective integer or floating-point literal multiplied by the imaginary unit *i*.

```
imaginary_lit = (decimal_digits | int_lit | float_lit) "i" .
```

For backward compatibility, an imaginary literal's integer part consisting entirely of decimal digits (and possibly underscores) is considered a decimal integer, even if it starts with a leading `0`.

```
0i
0123i         // == 123i for backward-compatibility
0o123i        // == 0o123 * 1i == 83i
0xabci        // == 0xabc * 1i == 2748i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
0x1p-2i       // == 0x1p-2 * 1i == 0.25i
```

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

Raw string literals are character sequences between back quotes, as in `` `foo` ``. Within the quotes, any character may appear except back quote. The value of a raw string literal is the string composed of the uninterpreted (implicitly UTF-8-encoded) characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines. Carriage return characters ('\r') inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear except newline and unescaped double quote. The text between the quotes forms the value of the literal, with backslash escapes interpreted as they are in rune literals (except that `\'` is illegal and `\"` is legal), with the same restrictions. The three-digit octal (`\`*nnn*) and two-digit hexadecimal (`\x`*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal `\377` and `\xFF` represent a single byte of

value `0xFF`=255,
while `ÿ`, `\u00FF`, `\U000000FF` and `\xc3\xbf` represent the two bytes `0xc3 0xbf` of the UTF-8 encoding of character U+00FF.

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
`abc`                // same as "abc"
`\n
\n`                  // same as "\\n\n\\n"
"\n"
"\""                 // same as `"`
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
"\uD800"             // illegal: surrogate half
"\U00110000"         // illegal: invalid Unicode code point
```

These examples all represent the same string:

```
"日本語"                                 // UTF-8 input text
`日本語`                                 // UTF-8 input text as a raw literal
"\u65e5\u672c\u8a9e"                    // the explicit Unicode code points
"\U000065e5\U0000672c\U00008a9e"        // the explicit Unicode code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"  // the explicit UTF-8 bytes
```

If the source code represents a character as two code points, such as a combining form involving an accent and a letter, the result will be an error if placed in a rune literal (it is not a single code point), and will appear as two code points if placed in a string literal.

Constants

There are *boolean constants*, *rune constants*, *integer constants*, *floating-point constants*, *complex constants*, and *string constants*. Rune, integer, floating-point, and complex constants are collectively called *numeric constants*.

A constant value is represented by a rune, integer, floating-point, imaginary, or string literal, an identifier denoting a constant, a constant expression, a conversion with a result that is a constant, or the result value of some built-in functions such as `min` or `max` applied to constant arguments, `unsafe.Sizeof` applied to certain values, `cap` or `len` applied to some expressions, `real` and `imag` applied to a complex constant and `complex` applied to numeric constants. The boolean truth values are represented by the predeclared constants `true` and `false`. The predeclared identifier iota denotes an integer constant.

In general, complex constants are a form of constant expression and are discussed in that section.

Numeric constants represent exact values of arbitrary precision and do not overflow. Consequently, there are no constants denoting the IEEE-754 negative zero, infinity, and not-a-number values.

Constants may be typed or *untyped*. Literal constants, `true`, `false`, `iota`, and certain constant expressions containing only untyped constant operands are untyped.

A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment statement or as an operand in an expression. It is an error if the constant value cannot be represented as a value of the respective type. If the type is a type parameter, the constant is converted into a non-constant value of the type parameter.

An untyped constant has a *default type* which is the type to which the constant is implicitly converted in contexts where a typed value is required, for instance, in a short variable declaration such as `i := 0` where there is no explicit type. The default type of an untyped constant is `bool`, `rune`, `int`, `float64`, `complex128`, or `string` respectively, depending on whether it is a boolean, rune, integer, floating-point, complex, or string constant.

Implementation restriction: Although numeric constants have arbitrary precision in the language, a compiler may implement them using an internal representation with limited precision. That said, every implementation must:

- Represent integer constants with at least 256 bits.
- Represent floating-point constants, including the parts of a complex constant, with a mantissa of at least 256 bits and a signed binary exponent of at least 16 bits.
- Give an error if unable to represent an integer constant precisely.
- Give an error if unable to represent a floating-point or complex constant due to overflow.
- Round to the nearest representable constant if unable to represent a floating-point or complex constant due to limits on precision.

These requirements apply both to literal constants and to the result of evaluating constant expressions.

Variables

A variable is a storage location for holding a *value*. The set of permissible values is determined by the variable's *type*.

A variable declaration or, for function parameters and results, the signature of a function declaration or function literal reserves storage for a named variable. Calling the built-in function `new` or taking the address of a composite literal allocates storage for a variable at run time. Such an anonymous variable is referred to via a (possibly implicit) pointer indirection.

Structured variables of array, slice, and struct types have elements and fields that may be addressed individually. Each such element acts like a variable.

The *static type* (or just *type*) of a variable is the type given in its declaration, the type provided in the `new` call or composite literal, or the type of an element of a structured variable. Variables of interface type also have a distinct *dynamic type*, which is the (non-interface) type of the value assigned to the variable at run time (unless the value is the predeclared identifier `nil`, which has no type). The dynamic type may vary during execution but values stored in interface variables are always assignable to the static type of the variable.

```
var x interface{}  // x is nil and has static type interface{}
var v *T           // v has value nil, static type *T
x = 42             // x has value 42 and dynamic type int
x = v              // x has value (*T)(nil) and dynamic type *T
```

A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable. If a variable has not yet been assigned a value, its value is the zero value for its type.

Types

A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a *type name*, if it has one, which must be followed by type arguments if the type is generic. A type may also be specified using a *type literal*, which composes a type from existing types.

```
Type      = TypeName [ TypeArgs ] | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeArgs  = "[" TypeList [ "," ] "]" .
TypeList  = Type { "," Type } .
TypeLit   = ArrayType | StructType | PointerType | FunctionType |
InterfaceType |
            SliceType | MapType | ChannelType .
```

Boolean types

A *boolean type* represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`; it is a defined type.

Numeric types

An *integer*, *floating-point*, or *complex* type represents the set of integer, floating-point, or complex values, respectively. They are collectively called *numeric types*. The predeclared architecture-independent numeric types are:

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to
4294967295)
uint64      the set of all unsigned 64-bit integers (0 to
18446744073709551615)

int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to
2147483647)
int64       the set of all signed 64-bit integers (-
9223372036854775808 to 9223372036854775807)

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers

complex64   the set of all complex numbers with float32 real and
imaginary parts
complex128  the set of all complex numbers with float64 real and
imaginary parts

byte        alias for uint8
rune        alias for int32
```

The value of an *n*-bit integer is *n* bits wide and represented using two's complement arithmetic.

There is also a set of predeclared integer types with implementation-specific sizes:

```
uint     either 32 or 64 bits
int      same size as uint
uintptr  an unsigned integer large enough to store the
uninterpreted bits of a pointer value
```

To avoid portability issues all numeric types are defined types and thus distinct except `byte`, which is an alias for `uint8`, and `rune`, which is an alias for `int32`. Explicit conversions are required when different numeric types are

mixed in an expression or assignment. For instance, `int32` and `int` are not the same type even though they may have the same size on a particular architecture.

String types

A *string type* represents the set of string values. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative. Strings are immutable: once created, it is impossible to change the contents of a string. The predeclared string type is `string`; it is a defined type.

The length of a string `s` can be discovered using the built-in function `len`. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through `len(s)-1`. It is illegal to take the address of such an element; if `s[i]` is the `i`'th byte of a string, `&s[i]` is invalid.

Array types

An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length of the array and is never negative.

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

The length is part of the array's type; it must evaluate to a non-negative constant representable by a value of type `int`. The length of array `a` can be discovered using the built-in function `len`. The elements can be addressed by integer indices 0 through `len(a)-1`. Array types are always one-dimensional but may be composed to form multi-dimensional types.

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

An array type `T` may not have an element of type `T`, or of a type containing `T` as a component, directly or indirectly, if those containing types are only array or struct types.

```
// invalid array types
type (
            T1 [10]T1                 // element type of T1 is T1
            T2 [10]struct{ f T2 }     // T2 contains T2 as component
of a struct
            T3 [10]T4                 // T3 contains T3 as component
of a struct in T4
            T4 struct{ f T3 }         // T4 contains T4 as component
of array T3 in a struct
)

// valid array types
type (
            T5 [10]*T5                // T5 contains T5 as component
of a pointer
            T6 [10]func() T6          // T6 contains T6 as component
of a function type
            T7 [10]struct{ f []T7 }   // T7 contains T7 as component
of a slice in a struct
)
```

Slice types

A slice is a descriptor for a contiguous segment of an *underlying array* and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The number of elements is called the length of the slice and is never negative. The value of an uninitialized slice is `nil`.

```
SliceType = "[" "]" ElementType .
```

The length of a slice `s` can be discovered by the built-in function `len`; unlike with arrays it may change during execution. The elements can be addressed by integer indices 0 through `len(s)-1`. The slice index of a given element may be less than the index of the same element in the underlying array.

A slice, once initialized, is always associated with an underlying array that holds its elements. A slice therefore shares storage with its array and with other slices of the same array; by contrast, distinct arrays always represent distinct storage.

Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. With arrays of arrays, the inner arrays are, by construction, always the same length; however with slices of slices (or arrays of slices), the inner lengths may vary dynamically. Moreover, the inner slices must be initialized individually.

Struct types

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly

(EmbeddedField). Within a struct, non-blank field names must be unique.

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type | EmbeddedField) [ Tag ] .
EmbeddedField = [ "*" ] TypeName [ TypeArgs ] .
Tag           = string_lit .
// An empty struct.
struct {}

// A struct with 6 fields.
struct {
            x, y int
            u float32
            _ float32  // padding
            A *[]int
            F func()
}
```

A field declared with a type but no explicit field name is called an *embedded field*. An embedded field must be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself may not be a pointer type. The unqualified type name acts as the field name.

```
// A struct with four embedded fields of types T1, *T2, P.T3 and
*P.T4
struct {
            T1        // field name is T1
            *T2       // field name is T2
            P.T3      // field name is T3
            *P.T4     // field name is T4
            x, y int  // field names are x and y
}
```

The following declaration is illegal because field names must be unique in a struct type:

```
struct {
            T     // conflicts with embedded field *T and *P.T
            *T    // conflicts with embedded field T and *P.T
            *P.T  // conflicts with embedded field T and *T
}
```

A field or method `f` of an embedded field in a struct `x` is called *promoted* if `x.f` is a legal selector that denotes that field or method `f`.

Promoted fields act like ordinary fields of a struct except that they cannot be used as field names in composite literals of the struct.

Given a struct type `S` and a named type `T`, promoted methods are included in the method set of the struct as follows:

- If `S` contains an embedded field `T`, the method sets of `S` and `*S` both include promoted methods with receiver `T`. The method set of `*S` also includes promoted methods with receiver `*T`.
- If `S` contains an embedded field `*T`, the method sets of `S` and `*S` both include promoted methods with receiver `T` or `*T`.

A field declaration may be followed by an optional string literal *tag*, which becomes an attribute for all the fields in the corresponding field declaration. An empty tag string is equivalent to an absent tag. The tags are made visible through a reflection interface and take part in type identity for structs but are otherwise ignored.

```
struct {
            x, y float64 ""  // an empty tag string is like an
absent tag
            name string  "any string is permitted as a tag"
            _    [4]byte "ceci n'est pas un champ de structure"
}

// A struct corresponding to a TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers;
// they follow the convention outlined by the reflect package.
struct {
            microsec  uint64 `protobuf:"1"`
            serverIP6 uint64 `protobuf:"2"`
}
```

A struct type `T` may not contain a field of type `T`, or of a type containing `T` as a component, directly or indirectly, if those containing types are only array or struct types.

```
// invalid struct types
type (
            T1 struct{ T1 }            // T1 contains a field of T1
            T2 struct{ f [10]T2 }      // T2 contains T2 as
component of an array
            T3 struct{ T4 }            // T3 contains T3 as
component of an array in struct T4
            T4 struct{ f [10]T3 }      // T4 contains T4 as
component of struct T3 in an array
)

// valid struct types
type (
            T5 struct{ f *T5 }         // T5 contains T5 as
component of a pointer
            T6 struct{ f func() T6 }   // T6 contains T6 as
component of a function type
            T7 struct{ f [10][]T7 }    // T7 contains T7 as
component of a slice in an array
)
```

Pointer types

A pointer type denotes the set of all pointers to variables of a given type, called the *base type* of the pointer. The value of an uninitialized pointer is `nil`.

```
PointerType = "*" BaseType .
BaseType    = Type .
*Point
*[4]int
```

Function types

A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is `nil`.

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

Within a list of parameters or results, the names (IdentifierList) must either all be present or all be absent. If present, each name stands for one item (parameter or result) of the specified type and all non-blank names in the signature must be unique. If absent, each type stands for one item of that type. Parameter and result lists are always parenthesized except that if there is exactly one unnamed result it may be written as an unparenthesized type.

The final incoming parameter in a function signature may have a type prefixed with …. A function with such a parameter is called *variadic* and may be invoked with zero or more arguments for that parameter.

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
func(prefix string, values ...int)
func(a, b int, z float64, opt ...interface{}) (success bool)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

Interface types

An interface type defines a *type set*. A variable of interface type can store a value of any type that is in the type set of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is `nil`.

```
InterfaceType  = "interface" "{" { InterfaceElem ";" } "}" .
InterfaceElem  = MethodElem | TypeElem .
MethodElem     = MethodName Signature .
MethodName     = identifier .
TypeElem       = TypeTerm { "|" TypeTerm } .
TypeTerm       = Type | UnderlyingType .
UnderlyingType = "~" Type .
```

An interface type is specified by a list of *interface elements*. An interface element is either a *method* or a *type element*, where a type element is a union of one or more *type terms*.

A type term is either a single type or a single underlying type.

Basic interfaces

In its most basic form an interface specifies a (possibly empty) list of methods. The type set defined by such an interface is the set of types which implement all of those methods, and the corresponding method set consists exactly of the methods specified by the interface. Interfaces whose type sets can be defined entirely by a list of methods are called *basic interfaces*.

```
// A simple File interface.
interface {
            Read([]byte) (int, error)
            Write([]byte) (int, error)
            Close() error
}
```

The name of each explicitly specified method must be unique and not blank.

```
interface {
            String() string
            String() string  // illegal: String not unique
            _(x int)         // illegal: method must have non-blank
name
}
```

More than one type may implement an interface. For instance, if two types `S1` and `S2` have the method set

```
func (p T) Read(p []byte) (n int, err error)
func (p T) Write(p []byte) (n int, err error)
func (p T) Close() error
```

(where `T` stands for either `S1` or `S2`) then the `File` interface is implemented by both `S1` and `S2`, regardless of what other methods `S1` and `S2` may have or share.

Every type that is a member of the type set of an interface implements that interface. Any given type may implement several distinct interfaces. For instance, all types implement the *empty interface* which stands for the set of all (non-interface) types:

```
interface{}
```

For convenience, the predeclared type `any` is an alias for the empty interface.

Similarly, consider this interface specification, which appears within a type declaration to define an interface called `Locker`:

```
type Locker interface {
            Lock()
            Unlock()
}
```

If `S1` and `S2` also implement

```
func (p T) Lock() { … }
func (p T) Unlock() { … }
```

they implement the `Locker` interface as well as the `File` interface.

Embedded interfaces

In a slightly more general form an interface `T` may use a (possibly qualified) interface type name `E` as an interface element. This is called *embedding* interface `E` in `T`. The type set of `T` is the *intersection* of the type sets defined by `T`'s explicitly declared methods and the type sets of `T`'s embedded interfaces. In other words, the type set of `T` is the set of all types that implement all the explicitly declared methods of `T` and also all the methods of `E`.

```
type Reader interface {
            Read(p []byte) (n int, err error)
            Close() error
}

type Writer interface {
            Write(p []byte) (n int, err error)
            Close() error
}

// ReadWriter's methods are Read, Write, and Close.
type ReadWriter interface {
            Reader  // includes methods of Reader in ReadWriter's
method set
            Writer  // includes methods of Writer in ReadWriter's
method set
}
```

General interfaces

In their most general form, an interface element may also be an arbitrary type term `T`, or a term of the form `~T` specifying the underlying type `T`, or a union of terms $t_1|t_2|…|t_n$. Together with method specifications, these elements enable the precise definition of an interface's type set as follows:

- The type set of the empty interface is the set of all non-interface types.
- The type set of a non-empty interface is the intersection of the type sets of its interface elements.
- The type set of a method specification is the set of all non-interface types whose method sets include that method.
- The type set of a non-interface type term is the set consisting of just that type.
- The type set of a term of the form `~T` is the set of all types whose underlying type is `T`.
- The type set of a *union* of terms $t_1|t_2|…|t_n$ is the union of the type sets of the terms.

By construction, an interface's type set never contains an interface type.

```
// An interface representing only the type int.
interface {
            int
}

// An interface representing all types with underlying type int.
interface {
            ~int
}

// An interface representing all types with underlying type int
that implement the String method.
interface {
            ~int
            String() string
}

// An interface representing an empty type set: there is no type
that is both an int and a string.
interface {
            int
            string
}
```

In a term of the form `~T`, the underlying type of `T` must be itself, and `T` cannot be an interface.

```
type MyInt int

interface {
            ~[]byte  // the underlying type of []byte is itself
            ~MyInt   // illegal: the underlying type of MyInt is not
MyInt
            ~error   // illegal: error is an interface
}
```

Union elements denote unions of type sets:

```
// The Float interface represents all floating-point types
// (including any named types whose underlying types are
// either float32 or float64).
type Float interface {
            ~float32 | ~float64
}
```

Implementing an interface

A type `T` implements an interface `I` if

- `T` is not an interface and is an element of the type set of `I`; or
- `T` is an interface and the type set of `T` is a subset of the type set of `I`.

A value of type `T` implements an interface if `T` implements the interface.

Map types

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique *keys* of another type, called the key type. The value of an uninitialized map is `nil`.

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

The comparison operators `==` and `!=` must be fully defined for operands of the key type; thus the key type must not be a function, map, or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

```
map[string]int
map[*T]struct{ x, y float64 }
map[string]interface{}
```

The number of map elements is called its length. For a map `m`, it can be discovered using the built-in function `len` and may change during execution. Elements may be added during execution using assignments and retrieved with index expressions; they may be removed with the `delete` and `clear` built-in function.

A new, empty map value is made using the built-in function `make`, which takes the map type and an optional capacity hint as arguments:

```
make(map[string]int)
make(map[string]int, 100)
```

The initial capacity does not bound its size: maps grow to accommodate the number of items stored in them, with the exception of `nil` maps. A `nil` map is equivalent to an empty map except that no elements may be added.

Blocks

A *block* is a possibly empty sequence of declarations and statements within matching brace brackets.

```
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
```

In addition to explicit blocks in the source code, there are implicit blocks:

1. The *universe block* encompasses all Go source text.
2. Each package has a *package block* containing all Go source text for that package.
3. Each file has a *file block* containing all Go source text in that file.
4. Each "if", "for", and "switch" statement is considered to be in its own implicit block.
5. Each clause in a "switch" or "select" statement acts as an implicit block.

Blocks nest and influence scoping.

Declarations and scope

A *declaration* binds a non-blank identifier to a constant, type, type parameter, variable, function, label, or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

The blank identifier may be used like any other identifier in a declaration, but it does not introduce a binding and thus is not declared. In the package block, the identifier `init` may only be used for `init` function declarations, and like the blank identifier it does not introduce a new binding.

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

The *scope* of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, label, or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.
3. The scope of the package name of an imported package is the file block of the file containing the import declaration.
4. The scope of an identifier denoting a method receiver, function parameter, or result variable is the function body.
5. The scope of an identifier denoting a type parameter of a function or declared by a method receiver begins after the name of the function and ends at the end of the function body.
6. The scope of an identifier denoting a type parameter of a type begins after the name of the type and ends at the end of the TypeSpec.
7. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block.
8. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope, it denotes the entity declared by the inner declaration.

The package clause is not a declaration; the package name does not appear in any scope. Its purpose is to identify the files belonging to the same package and to specify the default package name for import declarations.

Label scopes

Labels are declared by labeled statements and are used in the "break", "continue", and "goto" statements. It is illegal to define a label that is never used. In contrast to other identifiers, labels are not block scoped and do not conflict with identifiers that are not labels. The scope of a label is the body of the function in which it is declared and excludes the body of any nested function.

Blank identifier

The *blank identifier* is represented by the underscore character _. It serves as an anonymous placeholder instead of a regular (non-blank) identifier and has special meaning in declarations, as an operand, and in assignment statements.

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

```
Types:
            any bool byte comparable
            complex64 complex128 error float32 float64
            int int8 int16 int32 int64 rune string
            uint uint8 uint16 uint32 uint64 uintptr

Constants:
            true false iota

Zero value:
            nil

Functions:
            append cap clear close complex copy delete imag len
            make max min new panic print println real recover
```

Constant declarations

A constant declaration binds a list of identifiers (the names of the constants) to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions, and the *n*th identifier on the left is bound to the value of the *n*th expression on the right.

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

If the type is present, all constants take the type specified, and the expressions must be assignable to that type, which must not be a type parameter. If the type is omitted, the constants take the individual types of the corresponding expressions. If the expression values are untyped constants, the declared constants remain untyped and the constant identifiers denote the constant values. For instance, if the expression is a floating-point literal, the constant identifier denotes a floating-point constant, even if the literal's fractional part is zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
            size int64 = 1024
            eof        = -1  // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo", untyped
integer and string constants
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

Within a parenthesized `const` declaration list the expression list may be omitted from any but the first ConstSpec. Such an empty list is equivalent to the textual substitution of the first preceding non-empty expression list and its type if any. Omitting the list of expressions is therefore equivalent to repeating the previous list. The number of identifiers must be equal to the number of expressions in the previous list. Together with the `iota` constant generator this mechanism permits light-weight declaration of sequential values:

```
const (
            Sunday = iota
            Monday
            Tuesday
            Wednesday
            Thursday
            Friday
            Partyday
            numberOfDays  // this constant is not exported
)
```

Type declarations

A type declaration binds an identifier, the *type name*, to a type. Type declarations come in two forms: alias declarations and type definitions.

```
TypeDecl = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec = AliasDecl | TypeDef .
```

Alias declarations

An alias declaration binds an identifier to the given type.

```
AliasDecl = identifier "=" Type .
```

Within the scope of the identifier, it serves as an *alias* for the type.

```
type (
            nodeList = []*Node  // nodeList and []*Node are
identical types
            Polar    = polar    // Polar and polar denote identical
types
)
```

Type definitions

A type definition creates a new, distinct type with the same underlying type and operations as the given type and binds an identifier, the *type name*, to it.

```
TypeDef = identifier [ TypeParameters ] Type .
```

The new type is called a *defined type*. It is different from any other type, including the type it is created from.

```
type (
            Point struct{ x, y float64 }  // Point and struct{ x, y
float64 } are different types
            polar Point                   // polar and Point denote
different types
)

type TreeNode struct {
            left, right *TreeNode
            value any
}

type Block interface {
            BlockSize() int
            Encrypt(src, dst []byte)
            Decrypt(src, dst []byte)
}
```

A defined type may have methods associated with it. It does not inherit any methods bound to the given type, but the method set of an interface type or of elements of a composite type remains unchanged:

```
// A Mutex is a data type with two methods, Lock and Unlock.
type Mutex struct         { /* Mutex fields */ }
func (m *Mutex) Lock()    { /* Lock implementation */ }
func (m *Mutex) Unlock()  { /* Unlock implementation */ }

// NewMutex has the same composition as Mutex but its method set is
empty.
type NewMutex Mutex

// The method set of PtrMutex's underlying type *Mutex remains
unchanged,
// but the method set of PtrMutex is empty.
type PtrMutex *Mutex

// The method set of *PrintableMutex contains the methods
// Lock and Unlock bound to its embedded field Mutex.
type PrintableMutex struct {
            Mutex
}

// MyBlock is an interface type that has the same method set as
Block.
type MyBlock Block
```

Type definitions may be used to define different boolean, numeric, or string types and associate methods with them:

```
type TimeZone int

const (
            EST TimeZone = -(5 + iota)
            CST
            MST
            PST
)
```

```
func (tz TimeZone) String() string {
            return fmt.Sprintf("GMT%+dh", tz)
}
```

If the type definition specifies type parameters, the type name denotes a *generic type*. Generic types must be instantiated when they are used.

```
type List[T any] struct {
            next  *List[T]
            value T
}
```

In a type definition the given type cannot be a type parameter.

```
type T[P any] P    // illegal: P is a type parameter

func f[T any]() {
            type L T   // illegal: T is a type parameter declared by
the enclosing function
}
```

A generic type may also have methods associated with it. In this case, the method receivers must declare the same number of type parameters as present in the generic type definition.

```
// The method Len returns the number of elements in the linked list
l.
func (l *List[T]) Len() int  { … }
```

Variable declarations

A variable declaration creates one or more variables, binds corresponding identifiers to them, and gives each a type and an initial value.

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "="
ExpressionList ) .
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
            i       int
            u, v, s = 2.0, 3.0, "bar"
)
var re, im = complexSqrt(-1)
var _, found = entries[name]  // map lookup; only interested in
"found"
```

If a list of expressions is given, the variables are initialized with the expressions following the rules for assignment statements. Otherwise, each variable is initialized to its zero value.

If a type is present, each variable is given that type. Otherwise, each variable is given the type of the corresponding initialization value in the assignment. If that value is an untyped constant, it is first implicitly converted to its default type; if it is an untyped boolean value, it is first implicitly converted to type `bool`. The predeclared value `nil` cannot be used to initialize a variable with no explicit type.

```
var d = math.Sin(0.5)  // d is float64
var i = 42             // i is int
var t, ok = x.(T)      // t is T, ok is bool
var n = nil            // illegal
```

Implementation restriction: A compiler may make it illegal to declare a variable inside a function body if the variable is never used.

Short variable declarations

A *short variable declaration* uses the syntax:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

It is shorthand for a regular variable declaration with initializer expressions but no types:

```
"var" IdentifierList "=" ExpressionList .
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w, _ := os.Pipe()  // os.Pipe() returns a connected pair of
Files and an error, if any
_, y, _ := coord(p)   // coord() returns three values; only
interested in y coordinate
```

Unlike regular variable declarations, a short variable declaration may *redeclare* variables provided they were originally declared earlier in the same block (or the parameter lists if the block is the function body) with the same type, and at least one of the non-blank variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original. The non-blank variable names on the left side of `:=` must be unique.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
x, y, x := 1, 2, 3                        // illegal: x repeated on
left side of :=
```

Short variable declarations may appear only inside functions. In some contexts such as the initializers

for "if", "for", or "switch" statements, they can be used to declare local temporary variables.

Function declarations

A function declaration binds an identifier, the *function name*, to a function.

```
FunctionDecl = "func" FunctionName [ TypeParameters ] Signature [
FunctionBody ] .
FunctionName = identifier .
FunctionBody = Block .
```

If the function's signature declares result parameters, the function body's statement list must end in a terminating statement.

```
func IndexRune(s string, r rune) int {
            for i, c := range s {
                        if c == r {
                                    return i
                        }
            }
            // invalid: missing return statement
}
```

If the function declaration specifies type parameters, the function name denotes a *generic function*. A generic function must be instantiated before it can be called or used as a value.

```
func min[T ~int|~float64](x, y T) T {
            if x < y {
                        return x
            }
            return y
}
```

A function declaration without type parameters may omit the body. Such a declaration provides the signature for a function implemented outside Go, such as an assembly routine.

```
func flushICache(begin, end uintptr)  // implemented externally
```

Method declarations

A method is a function with a *receiver*. A method declaration binds an identifier, the *method name*, to a method, and associates the method with the receiver's *base type*.

```
MethodDecl = "func" Receiver MethodName Signature [ FunctionBody ]
.
Receiver   = Parameters .
```

The receiver is specified via an extra parameter section preceding the method name. That parameter section must declare a single non-variadic parameter, the receiver. Its type must be a defined type `T` or a pointer to a defined type `T`, possibly followed by a list of type parameter names `[P1, P2, …]` enclosed in square brackets. `T` is called the receiver *base type*. A receiver base type cannot be a pointer or interface type and it must be defined in the same package as the method. The method is said to be *bound* to its receiver base type and the method name is visible only within selectors for type `T` or `*T`.

Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

Operands

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly qualified) non-blank identifier denoting a constant, variable, or function, or a parenthesized expression.

```
Operand     = Literal | OperandName [ TypeArgs ] | "(" Expression
")" .
Literal     = BasicLit | CompositeLit | FunctionLit .
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit |
string_lit .
OperandName = identifier | QualifiedIdent .
```

An operand name denoting a generic function may be followed by a list of type arguments; the resulting operand is an instantiated function.

The blank identifier may appear as an operand only on the left-hand side of an assignment statement.

Implementation restriction: A compiler need not report an error if an operand's type is a type parameter with an empty type set. Functions with such type parameters cannot be instantiated; any attempt will lead to an error at the instantiation site.

Composite literals

Composite literals construct new composite values each time they are evaluated. They consist of the type of the literal followed by a brace-bound list of elements. Each

element may optionally be preceded by a corresponding key.

```
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType
|
                SliceType | MapType | TypeName [ TypeArgs ] .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .
```

The LiteralType's core type `T` must be a struct, array, slice, or map type (the syntax enforces this constraint except when the type is given as a TypeName). The types of the elements and keys must be assignable to the respective field, element, and key types of type `T`; there is no additional conversion. The key is interpreted as a field name for struct literals, an index for array and slice literals, and a key for map literals. For map literals, all elements must have a key. It is an error to specify multiple elements with the same field name or constant key value. For non-constant map keys, see the section on evaluation order.

For struct literals the following rules apply:

- A key must be a field name declared in the struct type.
- An element list that does not contain any keys must list an element for each struct field in the order in which the fields are declared.
- If any element has a key, every element must have a key.
- An element list that contains keys does not need to have an element for each struct field. Omitted fields get the zero value for that field.
- A literal may omit the element list; such a literal evaluates to the zero value for its type.
- It is an error to specify an element for a non-exported field of a struct belonging to a different package.

Given the declarations

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

one may write

```
origin := Point3D{}                            // zero value for
Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  // zero value for
line.q.x
```

For array and slice literals the following rules apply:

- Each element has an associated integer index marking its position in the array.
- An element with a key uses the key as its index. The key must be a non-negative constant representable by a value of type `int`; and if it is typed it must be of integer type.
- An element without a key uses the previous element's index plus one. If the first element has no key, its index is zero.

Taking the address of a composite literal generates a pointer to a unique variable initialized with the literal's value.

```
var pointer *Point3D = &Point3D{y: 1000}
```

Note that the zero value for a slice or map type is not the same as an initialized but empty value of the same type. Consequently, taking the address of an empty slice or map composite literal does not have the same effect as allocating a new slice or map value with new.

```
p1 := &[]int{}    // p1 points to an initialized, empty slice with
value []int{} and length 0
p2 := new([]int)  // p2 points to an uninitialized slice with value
nil and length 0
```

The length of an array literal is the length specified in the literal type. If fewer elements than the length are provided in the literal, the missing elements are set to the zero value for the array element type. It is an error to provide elements with index values outside the index range of the array. The notation … specifies an array length equal to the maximum element index plus one.

```
buffer := [10]string{}             // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}       // len(intSet) == 6
days := [...]string{"Sat", "Sun"}  // len(days) == 2
```

A slice literal describes the entire underlying array literal. Thus the length and capacity of a slice literal are the maximum element index plus one. A slice literal has the form

```
[]T{x1, x2, … xn}
```

and is shorthand for a slice operation applied to an array:

```
tmp := [n]T{x1, x2, … xn}
tmp[0 : n]
```

Within a composite literal of array, slice, or map type T, elements or map keys that are themselves composite literals may elide the respective literal type if it is identical to the element or key type of T. Similarly, elements or keys that are addresses of composite literals may elide the &T when the element or key type is *T.

```
[...]Point{{1.5, -3.5}, {0, 0}}     // same as
[...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}          // same as [][]int{[]int{1, 2,
3}, []int{4, 5}}
[][]Point{{{0, 1}, {1, 2}}}         // same as
[][]Point{[]Point{Point{0, 1}, Point{1, 2}}}
map[string]Point{"orig": {0, 0}}    // same as
map[string]Point{"orig": Point{0, 0}}
map[Point]string{{0, 0}: "orig"}    // same as
map[Point]string{Point{0, 0}: "orig"}

type PPoint *Point
[2]*Point{{1.5, -3.5}, {}}          // same as
[2]*Point{&Point{1.5, -3.5}, &Point{}}
[2]PPoint{{1.5, -3.5}, {}}          // same as
[2]PPoint{PPoint(&Point{1.5, -3.5}), PPoint(&Point{})}
```

A parsing ambiguity arises when a composite literal using the TypeName form of the LiteralType appears as an operand between the keyword and the opening brace of the block of an "if", "for", or "switch" statement, and the composite literal is not enclosed in parentheses, square brackets, or curly braces. In this rare case, the opening brace of the literal is erroneously parsed as the one introducing the block of statements. To resolve the ambiguity, the composite literal must appear within parentheses.

```
if x == (T{a,b,c}[i]) { … }
if (x == T{a,b,c}[i]) { … }
```

Examples of valid array, slice, and map literals:

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 2147483647}

// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true,
'u': true, 'y': true}

// the array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
            "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
            "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Function literals

A function literal represents an anonymous function. Function literals cannot declare type parameters.

```
FunctionLit = "func" Signature FunctionBody .
func(a, b int, z float64) bool { return a*b < int(z) }
```

A function literal can be assigned to a variable or invoked directly.

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK }(replyChan)
```

Function literals are *closures*: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

Primary expressions

Primary expressions are the operands for unary and binary expressions.

```
PrimaryExpr =
            Operand |
            Conversion |
            MethodExpr |
            PrimaryExpr Selector |
            PrimaryExpr Index |
            PrimaryExpr Slice |
            PrimaryExpr TypeAssertion |
            PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression [ "," ] "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
                 "[" [ Expression ] ":" Expression ":" Expression
"]" .
TypeAssertion  = "." "(" Type ")" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList
] ) [ "..." ] [ "," ] ] ")" .
x
2
(s + ".txt")
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
f.p[i].x()
```

Method expressions

If M is in the method set of type T, T.M is a function that is callable as a regular function with the same arguments as M prefixed by an additional argument that is the receiver of the method.

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = Type .
```

Consider a struct type `T` with two methods, `Mv`, whose receiver is of type `T`, and `Mp`, whose receiver is of type `*T`.

```
type T struct {
            a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer
receiver

var t T
```

It is legal to derive a function value from a method of an interface type. The resulting function takes an explicit receiver of that interface type.

Method values

If the expression `x` has static type `T` and `M` is in the method set of type `T`, `x.M` is called a *method value*. The method value `x.M` is a function value that is callable with the same arguments as a method call of `x.M`. The expression `x` is evaluated and saved during the evaluation of the method value; the saved copy is then used as the receiver in any calls, which may be executed later.

```
type S struct { *T }
type T int
func (t T) M() { print(t) }

t := new(T)
s := S{T: t}
f := t.M                    // receiver *t is evaluated and stored
in f
g := s.M                    // receiver *(s.T) is evaluated and
stored in g
*t = 42                     // does not affect stored receivers in
f and g
```

The type `T` may be an interface or non-interface type.

As in the discussion of method expressions above, consider a struct type `T` with two methods, `Mv`, whose receiver is of type `T`, and `Mp`, whose receiver is of type `*T`.

```
type T struct {
            a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer
receiver

var t T
var pt *T
func makeT() T
```

Index expressions

A primary expression of the form

```
a[x]
```

denotes the element of the array, pointer to array, slice, string or map `a` indexed by `x`. The value `x` is called the *index* or *map key*, respectively. The following rules apply:

If `a` is neither a map nor a type parameter:

- the index `x` must be an untyped constant or its core type must be an integer
- a constant index must be non-negative and representable by a value of type `int`
- a constant index that is untyped is given type `int`
- the index `x` is *in range* if `0 <= x < len(a)`, otherwise it is *out of range*

For `a` of array type `A`:

- a constant index must be in range
- if `x` is out of range at run time, a run-time panic occurs
- `a[x]` is the array element at index `x` and the type of `a[x]` is the element type of `A`

For `a` of pointer to array type:

- `a[x]` is shorthand for `(*a)[x]`

For `a` of slice type `S`:

- if `x` is out of range at run time, a run-time panic occurs
- `a[x]` is the slice element at index `x` and the type of `a[x]` is the element type of `S`

For `a` of string type:

- a constant index must be in range if the string `a` is also constant
- if `x` is out of range at run time, a run-time panic occurs
- `a[x]` is the non-constant byte value at index `x` and the type of `a[x]` is `byte`
- `a[x]` may not be assigned to

For `a` of map type `M`:

- `x`'s type must be assignable to the key type of `M`
- if the map contains an entry with key `x`, `a[x]` is the map element with key `x` and the type of `a[x]` is the element type of `M`
- if the map is `nil` or does not contain such an entry, `a[x]` is the zero value for the element type of `M`

For `a` of type parameter type `P`:

- The index expression `a[x]` must be valid for values of all types in `P`'s type set.
- The element types of all types in `P`'s type set must be identical. In this context, the element type of a string type is `byte`.
- If there is a map type in the type set of `P`, all types in that type set must be map types, and the respective key types must be all identical.
- `a[x]` is the array, slice, or string element at index `x`, or the map element with key `x` of the type argument that `P` is instantiated with, and the type of `a[x]` is the type of the (identical) element types.
- `a[x]` may not be assigned to if `P`'s type set includes string types.

Otherwise `a[x]` is illegal.

An index expression on a map `a` of type `map[K]V` used in an assignment statement or initialization of the special form

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

yields an additional untyped boolean value. The value of `ok` is `true` if the key `x` is present in the map, and `false` otherwise.

Assigning to an element of a `nil` map causes a run-time panic.

Slice expressions

Slice expressions construct a substring or slice from a string, array, pointer to array, or slice. There are two variants: a simple form that specifies a low and high bound, and a full form that also specifies a bound on the capacity.

Simple slice expressions

The primary expression

```
a[low : high]
```

constructs a substring or slice. The core type of `a` must be a string, array, pointer to array, slice, or a `bytestring`. The *indices* `low` and `high` select which elements of operand `a` appear in the result. The result has indices starting at 0 and length equal to `high - low`. After slicing the array `a`

```
a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]
```

the slice `s` has type `[]int`, length 3, capacity 4, and elements

```
s[0] == 2
s[1] == 3
s[2] == 4
```

For convenience, any of the indices may be omitted. A missing `low` index defaults to zero; a missing `high` index defaults to the length of the sliced operand:

```
a[2:]  // same as a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

If `a` is a pointer to an array, `a[low : high]` is shorthand for `(*a)[low : high]`.

For arrays or strings, the indices are *in range* if `0 <= low <= high <= len(a)`, otherwise they are *out of range*. For slices, the upper index bound is the slice capacity `cap(a)` rather than the length. A constant index must be non-negative and representable by a value of type `int`; for arrays or constant strings, constant indices must also be in range. If both indices are constant, they must satisfy `low <= high`. If the indices are out of range at run time, a run-time panic occurs.

Except for untyped strings, if the sliced operand is a string or slice, the result of the slice operation is a non-constant value of the same type as the operand. For untyped string operands the result is a non-constant value of type `string`. If the sliced operand is an array, it must be addressable and the result of the slice operation is a slice with the same element type as the array.

If the sliced operand of a valid slice expression is a `nil` slice, the result is a `nil` slice. Otherwise, if the result is a slice, it shares its underlying array with the operand.

```
var a [10]int
s1 := a[3:7]   // underlying array of s1 is array a; &s1[2] ==
&a[5]
s2 := s1[1:4]  // underlying array of s2 is underlying array of s1
which is array a; &s2[1] == &a[5]
s2[1] = 42     // s2[1] == s1[2] == a[5] == 42; they all refer to
the same underlying array element

var s []int
s3 := s[:0]    // s3 == nil
```

Full slice expressions

The primary expression

```
a[low : high : max]
```

constructs a slice of the same type, and with the same length and elements as the simple slice expression `a[low : high]`. Additionally, it controls the resulting slice's capacity by setting it to `max - low`. Only the first index may be omitted; it defaults to 0. The core type of `a` must be an array, pointer to array, or slice (but not a string). After slicing the array `a`

```
a := [5]int{1, 2, 3, 4, 5}
t := a[1:3:5]
```

the slice `t` has type `[]int`, length 2, capacity 4, and elements

```
t[0] == 2
t[1] == 3
```

As for simple slice expressions, if `a` is a pointer to an array, `a[low : high : max]` is shorthand for `(*a)[low : high : max]`. If the sliced operand is an array, it must be addressable.

The indices are *in range* if `0 <= low <= high <= max <= cap(a)`, otherwise they are *out of range*. A constant index must be non-negative and representable by a value of type `int`; for arrays, constant indices must also be in range. If multiple indices are constant, the constants that are present must be in range relative to each other. If the indices are out of range at run time, a run-time panic occurs.

Operators

Operators combine operands into expressions.

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

The right operand in a shift expression must have integer type or be an untyped constant representable by a value of type `uint`. If the left operand of a non-constant shift expression is an untyped constant, it is first implicitly converted to the type it would assume if the shift expression were replaced by its left operand alone.

```
var a [1024]byte
var s uint = 33

// The results of the following examples are given for 64-bit ints.
var i = 1<<s                  // 1 has type int
var j int32 = 1<<s            // 1 has type int32; j == 0
var k = uint64(1<<s)          // 1 has type uint64; k == 1<<33
var m int = 1.0<<s            // 1.0 has type int; m == 1<<33
var n = 1.0<<s == j           // 1.0 has type int32; n == true
var o = 1<<s == 2<<s          // 1 and 2 have type int; o == false
var p = 1<<s == 1<<33         // 1 has type int; p == true
var u = 1.0<<s                // illegal: 1.0 has type float64,
cannot shift
var u1 = 1.0<<s != 0          // illegal: 1.0 has type float64,
cannot shift
var u2 = 1<<s != 1.0          // illegal: 1 has type float64,
cannot shift
var v1 float32 = 1<<s         // illegal: 1 has type float32,
cannot shift
var v2 = string(1<<s)         // illegal: 1 is converted to a
string, cannot shift
var w int64 = 1.0<<33         // 1.0<<33 is a constant shift
expression; w == 1<<33
var x = a[1.0<<s]             // panics: 1.0 has type int, but
1<<33 overflows array bounds
var b = make([]byte, 1.0<<s)  // 1.0 has type int; len(b) == 1<<33

// The results of the following examples are given for 32-bit ints,
// which means the shifts will overflow.
var mm int = 1.0<<s           // 1.0 has type int; mm == 0
var oo = 1<<s == 2<<s         // 1 and 2 have type int; oo == true
var pp = 1<<s == 1<<33        // illegal: 1 has type int, but
1<<33 overflows int
var xx = a[1.0<<s]            // 1.0 has type int; xx == a[0]
var bb = make([]byte, 1.0<<s) // 1.0 has type int; len(bb) == 0
```

Operator precedence

Unary operators have the highest precedence. As the `++` and `--` operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement `*p++` is the same as `(*p)++`.

There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, `&&` (logical AND), and finally `||` (logical OR):

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Binary operators of the same precedence associate from left to right. For instance, `x / y * z` is the same as `(x / y) * z`.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chanInt > 0
```

Arithmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (`+`, `-`, `*`, `/`) apply to integer, floating-point, and complex types; `+` also applies to strings. The bitwise logical and shift operators apply to integers only.

```
+    sum                    integers, floats, complex values,
strings
-    difference             integers, floats, complex values
*    product                integers, floats, complex values
/    quotient               integers, floats, complex values
%    remainder              integers

&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << integer >= 0
>>   right shift            integer >> integer >= 0
```

Integer operators

For two integer values `x` and `y`, the integer quotient `q = x / y` and remainder `r = x % y` satisfy the following relationships:

```
x = q*y + r  and  |r| < |y|
```

with `x / y` truncated towards zero ("truncated division").

```
 x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

The one exception to this rule is that if the dividend `x` is the most negative value for the int type of `x`, the quotient `q = x / -1` is equal to `x` (and `r = 0`) due to two's-complement integer overflow:

```
                         x, q
int8                      -128
int16                   -32768
int32              -2147483648
int64    -9223372036854775808
```

If the divisor is a constant, it must not be zero. If the divisor is zero at run time, a run-time panic occurs. If the dividend is non-negative and the divisor is a constant power of 2, the division may be replaced by a right shift, and computing the remainder may be replaced by a bitwise AND operation:

```
 x     x / 4     x % 4     x >> 2     x & 3
 11      2         3         2          3
-11     -2        -3        -3          1
```

The shift operators shift the left operand by the shift count specified by the right operand, which must be non-negative. If the shift count is negative at run time, a run-time panic occurs. The shift operators implement arithmetic shifts if the left operand is a signed integer and logical shifts if it is an unsigned integer. There is no upper limit on the shift count. Shifts behave as if the left operand is shifted `n` times by 1 for a shift count of `n`. As a result, `x << 1` is the same as `x*2` and `x >> 1` is the same as `x/2` but truncated towards negative infinity.

For integer operands, the unary operators `+`, `-`, and `^` are defined as follows:

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits set to 1"
for unsigned x
                                        and  m = -1 for signed x
```

Integer overflow

For unsigned integer values, the operations `+`, `-`, `*`, and `<<` are computed modulo 2^n, where *n* is the bit width of the unsigned integer's type. Loosely speaking, these unsigned integer operations discard high bits upon overflow, and programs may rely on "wrap around".

For signed integers, the operations `+`, `-`, `*`, `/`, and `<<` may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation, the operation, and its operands. Overflow does not cause a run-time panic. A compiler may not

optimize code under the assumption that overflow does not occur. For instance, it may not assume that `x < x + 1` is always true.

Floating-point operators

For floating-point and complex numbers, `+x` is the same as `x`, while `-x` is the negation of `x`. The result of a floating-point or complex division by zero is not specified beyond the IEEE-754 standard; whether a run-time panic occurs is implementation-specific.

String concatenation

Strings can be concatenated using the `+` operator or the `+=` assignment operator:

```
s := "hi" + string(c)
s += " and good bye"
```

String addition creates a new string by concatenating the operands.

Comparison operators

Comparison operators compare two operands and yield an untyped boolean value.

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

The equality operators `==` and `!=` apply to operands of *comparable* types. The ordering operators `<`, `<=`, `>`, and `>=` apply to operands of *ordered* types. These terms and the result of the comparisons are defined as follows:

- Boolean types are comparable. Two boolean values are equal if they are either both `true` or both `false`.
- Integer types are comparable and ordered. Two integer values are compared in the usual way.
- Floating-point types are comparable and ordered. Two floating-point values are compared as defined by the IEEE-754 standard.
- Complex types are comparable. Two complex values `u` and `v` are equal if both `real(u) == real(v)` and `imag(u) == imag(v)`.
- String types are comparable and ordered. Two string values are compared lexically byte-wise.
- Pointer types are comparable. Two pointer values are equal if they point to the same variable or if both have value `nil`. Pointers to distinct zero-size variables may or may not be equal.
- Channel types are comparable. Two channel values are equal if they were created by the same call to `make` or if both have value `nil`.
- Interface types that are not type parameters are comparable. Two interface values are equal if they have identical dynamic types and equal dynamic values or if both have value `nil`.
- A value `x` of non-interface type `X` and a value `t` of interface type `T` can be compared if type `X` is comparable and `X` implements `T`. They are equal if `t`'s dynamic type is identical to `X` and `t`'s dynamic value is equal to `x`.
- Struct types are comparable if all their field types are comparable. Two struct values are equal if their corresponding non-blank field values are equal. The fields are compared in source order, and comparison stops as soon as two field values differ (or all fields have been compared).
- Array types are comparable if their array element types are comparable. Two array values are equal if their corresponding element values are equal. The elements are compared in ascending index order, and comparison stops as soon as two element values differ (or all elements have been compared).
- Type parameters are comparable if they are strictly comparable (see below).

A comparison of two interface values with identical dynamic types causes a run-time panic if that type is not comparable. This behavior applies not only to direct interface value comparisons but also when comparing arrays of interface values or structs with interface-valued fields.

Slice, map, and function types are not comparable. However, as a special case, a slice, map, or function value may be compared to the predeclared identifier `nil`. Comparison of pointer, channel, and interface values

to `nil` is also allowed and follows from the general rules above.

```
const c = 3 < 4            // c is the untyped boolean constant
true

type MyBool bool
var x, y int
var (
            // The result of a comparison is an untyped boolean.
            // The usual assignment rules apply.
            b3        = x == y // b3 has type bool
            b4 bool   = x == y // b4 has type bool
            b5 MyBool = x == y // b5 has type MyBool
)
```

A type is *strictly comparable* if it is comparable and not an interface type nor composed of interface types. Specifically:

- Boolean, numeric, string, pointer, and channel types are strictly comparable.
- Struct types are strictly comparable if all their field types are strictly comparable.
- Array types are strictly comparable if their array element types are strictly comparable.
- Type parameters are strictly comparable if all types in their type set are strictly comparable.

Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

Conversions

A conversion changes the type of an expression to the type specified by the conversion. A conversion may appear literally in the source, or it may be *implied* by the context in which an expression appears.

An *explicit* conversion is an expression of the form `T(x)` where `T` is a type and `x` is an expression that can be converted to type `T`.

```
Conversion = Type "(" Expression [ "," ] ")" .
```

If the type starts with the operator `*` or `<-`, or if the type starts with the keyword `func` and has no result list, it must be parenthesized when necessary to avoid ambiguity:

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to *Point
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to <-chan int
func()(x)        // function signature func() x
(func())(x)      // x is converted to func()
(func() int)(x)  // x is converted to func() int
func() int(x)    // x is converted to func() int (unambiguous)
```

A constant value `x` can be converted to type `T` if `x` is representable by a value of `T`. As a special case, an integer constant `x` can be explicitly converted to a string type using the same rule as for non-constant `x`.

Converting a constant to a type that is not a type parameter yields a typed constant.

```
uint(iota)               // iota value of type uint
float32(2.718281828)     // 2.718281828 of type float32
complex128(1)            // 1.0 + 0.0i of type complex128
float32(0.49999999)      // 0.5 of type float32
float64(-1e-1000)        // 0.0 of type float64
string('x')              // "x" of type string
string(0x266c)           // "♬" of type string
myString("foo" + "bar")  // "foobar" of type myString
string([]byte{'a'})      // not a constant: []byte{'a'} is not a
constant
(*int)(nil)              // not a constant: nil is not a constant,
*int is not a boolean, numeric, or string type
int(1.2)                 // illegal: 1.2 cannot be represented as
an int
string(65.0)             // illegal: 65.0 is not an integer
constant
```

Conversions between numeric types

For the conversion of non-constant numeric values, the following rules apply:

1. When converting between integer types, if the value is a signed integer, it is sign extended to implicit infinite precision; otherwise it is zero extended. It is then truncated to fit in the result type's size. For example, if `v := uint16(0x10F0)`, then `uint32(int8(v)) == 0xFFFFFFF0`. The conversion always yields a valid value; there is no indication of overflow.
2. When converting a floating-point number to an integer, the fraction is discarded (truncation towards zero).
3. When converting an integer or floating-point number to a floating-point type, or a complex number to another complex type, the result value is rounded to the precision specified by the destination type. For instance, the value of a variable `x` of type `float32` may be stored using additional precision beyond that of an IEEE-754

32-bit number, but float32(x) represents the result of rounding `x`'s value to 32-bit precision. Similarly, `x + 0.1` may use more than 32 bits of precision, but `float32(x + 0.1)` does not.

In all non-constant conversions involving floating-point or complex values, if the result type cannot represent the value the conversion succeeds but the result value is implementation-dependent.

Conversions to and from a string type

Converting a slice of bytes to a string type yields a string whose successive bytes are the elements of the slice.

```
1.  string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})   //
    "hellø"
2.  string([]byte{})                                     // ""
3.  string([]byte(nil))                                  // ""
4.
5.  type bytes []byte
6.  string(bytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})    //
    "hellø"
7.
8.  type myByte byte
9.  string([]myByte{'w', 'o', 'r', 'l', 'd', '!'})       //
    "world!"
10. myString([]myByte{'\xf0', '\x9f', '\x8c', '\x8d'})   //
    "🌍"
```

Converting a slice of runes to a string type yields a string that is the concatenation of the individual rune values converted to strings.

```
11. string([]rune{0x767d, 0x9d6c, 0x7fd4})   //
    "\u767d\u9d6c\u7fd4" == "白鵬翔"
12. string([]rune{})                         // ""
13. string([]rune(nil))                      // ""
14.
15. type runes []rune
16. string(runes{0x767d, 0x9d6c, 0x7fd4})    //
    "\u767d\u9d6c\u7fd4" == "白鵬翔"
17.
18. type myRune rune
19. string([]myRune{0x266b, 0x266c})         // "\u266b\u266c"
    == "♫♬"
20. myString([]myRune{0x1f30e})              // "\U0001f30e"
    == "🌎"
```

Converting a value of a string type to a slice of bytes type yields a slice whose successive elements are the bytes of the string.

```
21. []byte("hellø")             // []byte{'h', 'e', 'l', 'l',
    '\xc3', '\xb8'}
22. []byte("")                  // []byte{}
23.
24. bytes("hellø")              // []byte{'h', 'e', 'l', 'l',
    '\xc3', '\xb8'}
25.
26. []myByte("world!")          // []myByte{'w', 'o', 'r',
    'l', 'd', '!'}
27. []myByte(myString("🌎"))    // []myByte{'\xf0', '\x9f',
    '\x8c', '\x8f'}
```

Converting a value of a string type to a slice of runes type yields a slice containing the individual Unicode code points of the string.

```
28. []rune(myString("白鵬翔"))   // []rune{0x767d, 0x9d6c,
    0x7fd4}
29. []rune("")                  // []rune{}
30.
31. runes("白鵬翔")              // []rune{0x767d, 0x9d6c,
    0x7fd4}
32.
33. []myRune("♫♬")              // []myRune{0x266b, 0x266c}
34. []myRune(myString("🍥"))    // []myRune{0x1f310}
```

Finally, for historical reasons, an integer value may be converted to a string type. This form of conversion yields a string containing the (possibly multi-byte) UTF-8 representation of the Unicode code point with the given integer value. Values outside the range of valid Unicode code points are converted to `"\uFFFD"`.

```
35. string('a')          // "a"
36. string(65)           // "A"
37. string('\xf8')       // "\u00f8" == "ø" == "\xc3\xb8"
38. string(-1)           // "\ufffd" == "\xef\xbf\xbd"
39.
40. type myString string
41. myString('\u65e5')   // "\u65e5" == "日" == "\xe6\x97\xa5"
```

Note: This form of conversion may eventually be removed from the language. The `go vet` tool flags certain integer-to-string conversions as potential errors. Library functions such as `utf8.AppendRune` or `utf8.EncodeRune` should be used instead.

Conversions from slice to array or array pointer

Converting a slice to an array yields an array containing the elements of the underlying array of the slice. Similarly, converting a slice to an array pointer yields a pointer to the underlying array of the slice. In both cases, if the length of the slice is less than the length of the array, a run-time panic occurs.

```
s := make([]byte, 2, 4)

a0 := [0]byte(s)
a1 := [1]byte(s[1:])     // a1[0] == s[1]
a2 := [2]byte(s)         // a2[0] == s[0]
a4 := [4]byte(s)         // panics: len([4]byte) > len(s)

s0 := (*[0]byte)(s)      // s0 != nil
s1 := (*[1]byte)(s[1:])  // &s1[0] == &s[1]
s2 := (*[2]byte)(s)      // &s2[0] == &s[0]
s4 := (*[4]byte)(s)      // panics: len([4]byte) > len(s)
```

```
var t []string
t0 := [0]string(t)       // ok for nil slice t
t1 := (*[0]string)(t)    // t1 == nil
t2 := (*[1]string)(t)    // panics: len([1]string) > len(t)

u := make([]byte, 0)
u0 := (*[0]byte)(u)      // u0 != nil
```

Constant expressions

Constant expressions may contain only constant operands and are evaluated at compile time.

Untyped boolean, numeric, and string constants may be used as operands wherever it is legal to use an operand of boolean, numeric, or string type, respectively.

A constant comparison always yields an untyped boolean constant. If the left operand of a constant shift expression is an untyped constant, the result is an integer constant; otherwise it is a constant of the same type as the left operand, which must be of integer type.

Any other operation on untyped constants results in an untyped constant of the same kind; that is, a boolean, integer, floating-point, complex, or string constant. If the untyped operands of a binary operation (other than a shift) are of different kinds, the result is of the operand's kind that appears later in this list: integer, rune, floating-point, complex. For example, an untyped integer constant divided by an untyped complex constant yields an untyped complex constant.

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point
constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point
constant)
const Θ float64 = 3/2      // Θ == 1.0   (type float64, 3/2 is
integer division)
const Π float64 = 3/2.     // Π == 1.5   (type float64, 3/2. is
float division)
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // illegal    (constant 8589934592
overflows int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed
floating-point constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

Applying the built-in function `complex` to untyped integer, rune, or floating-point constants yields an untyped complex constant.

```
const ic = complex(0, c)   // ic == 3.75i  (untyped complex
constant)
const i0 = complex(0, 0)   // i0 == 1i     (type complex128)
```

Constant expressions are always evaluated exactly; intermediate values and the constants themselves may require precision significantly larger than supported by any predeclared type in the language. The following are legal declarations:

```
const Huge = 1 << 100         // Huge ==
1267650600228229401496703205376  (untyped integer constant)
const Four int8 = Huge >> 98  // Four == 4
(type int8)
```

The divisor of a constant division or remainder operation must not be zero:

```
3.14 / 0.0   // illegal: division by zero
```

The values of *typed* constants must always be accurately representable by values of the constant type. The following constant expressions are illegal:

```
uint(-1)     // -1 cannot be represented as a uint
int(3.14)    // 3.14 cannot be represented as an int
int64(Huge)  // 1267650600228229401496703205376 cannot be
represented as an int64
Four * 300   // operand 300 cannot be represented as an int8 (type
of Four)
Four * 100   // product 400 cannot be represented as an int8 (type
of Four)
```

The mask used by the unary bitwise complement operator `^` matches the rule for non-constants: the mask is all 1s for unsigned constants and -1 for signed and untyped constants.

```
^1         // untyped integer constant, equal to -2
uint8(^1)  // illegal: same as uint8(-2), -2 cannot be represented
as a uint8
^uint8(1)  // typed uint8 constant, same as 0xFF ^ uint8(1) =
uint8(0xFE)
int8(^1)   // same as int8(-2)
^int8(1)   // same as -1 ^ int8(1) = -2
```

Implementation restriction: A compiler may use rounding while computing untyped floating-point or complex constant expressions; see the implementation restriction in the section on constants. This rounding may cause a floating-point constant expression to be invalid in an integer context, even if it would be integral when calculated using infinite precision, and vice versa.

Statements

Statements control execution.

```
Statement =
            Declaration | LabeledStmt | SimpleStmt |
            GoStmt | ReturnStmt | BreakStmt | ContinueStmt |
GotoStmt |
            FallthroughStmt | Block | IfStmt | SwitchStmt |
SelectStmt | ForStmt |
            DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt |
Assignment | ShortVarDecl .
```

Terminating statements

A *terminating statement* interrupts the regular flow of control in a block. The following statements are terminating:

1. A "return" or "goto" statement.
2. A call to the built-in function `panic`.
3. A block in which the statement list ends in a terminating statement.
4. An "if" statement in which:
    - the "else" branch is present, and
    - both branches are terminating statements.
5. A "for" statement in which:
    - there are no "break" statements referring to the "for" statement, and
    - the loop condition is absent, and
    - the "for" statement does not use a range clause.
6. A "switch" statement in which:
    - there are no "break" statements referring to the "switch" statement,
    - there is a default case, and
    - the statement lists in each case, including the default, end in a terminating statement, or a possibly labeled "fallthrough" statement.
7. A "select" statement in which:
    - there are no "break" statements referring to the "select" statement, and
    - the statement lists in each case, including the default if present, end in a terminating statement.
8. A labeled statement labeling a terminating statement.

All other statements are not terminating.

A statement list ends in a terminating statement if the list is not empty and its final non-empty statement is terminating.

Empty statements

The empty statement does nothing.

```
EmptyStmt = .
```

Labeled statements

A labeled statement may be the target of a `goto`, `break` or `continue` statement.

```
LabeledStmt = Label ":" Statement .
Label       = identifier .
Error: log.Panic("error encountered")
```

Expression statements

With the exception of specific built-in functions, function and method calls and receive operations can appear in statement context. Such statements may be parenthesized.

```
ExpressionStmt = Expression .
```

The following built-in functions are not permitted in statement context:

```
append cap complex imag len make new real
unsafe.Add unsafe.Alignof unsafe.Offsetof unsafe.Sizeof
unsafe.Slice unsafe.SliceData unsafe.String unsafe.StringData
h(x+y)
f.Close()
<-ch
(<-ch)
len("foo")  // illegal if len is the built-in function
```

IncDec statements

The "++" and "--" statements increment or decrement their operands by the untyped constant `1`. As with an assignment, the operand must be addressable or a map index expression.

```
IncDecStmt = Expression ( "++" | "--" ) .
```

The following assignment statements are semantically equivalent:

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

Assignment statements

An *assignment* replaces the current value stored in a variable with a new value specified by an expression. An assignment statement may assign a single value to a single variable, or multiple values to a matching number of variables.

```
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .
```

Each left-hand side operand must be addressable, a map index expression, or (for `=` assignments only) the blank identifier. Operands may be parenthesized.

```
x = 1
*p = f()
a[i] = 23
(k) = <-ch  // same as: k = <-ch
```

An *assignment operation* `x` *op*`=` `y` where *op* is a binary arithmetic operator is equivalent to `x = x` *op* `(y)` but evaluates `x` only once. The *op*`=` construct is a single token. In assignment operations, both the left- and right-hand expression lists must contain exactly one single-valued expression, and the left-hand expression must not be the blank identifier.

```
a[i] <<= 2
i &^= 1<<n
```

A tuple assignment assigns the individual elements of a multi-valued operation to a list of variables. There are two forms. In the first, the right hand operand is a single multi-valued expression such as a function call, a channel or map operation, or a type assertion. The number of operands on the left hand side must match the number of values. For instance, if `f` is a function returning two values,

```
x, y = f()
```

assigns the first value to `x` and the second to `y`. In the second form, the number of operands on the left must equal the number of expressions on the right, each of which must be single-valued, and the *n*th expression on the right is assigned to the *n*th operand on the left:

```
one, two, three = '一', '二', '三'
```

The blank identifier provides a way to ignore right-hand side values in an assignment:

```
_ = x       // evaluate x but ignore it
x, _ = f()  // evaluate f() but ignore second result value
```

The assignment proceeds in two phases. First, the operands of index expressions and pointer indirections (including implicit pointer indirections in selectors) on the left and the expressions on the right are all evaluated in the usual order. Second, the assignments are carried out in left-to-right order.

```
a, b = b, a  // exchange a and b

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
            break
}
// after this loop, i == 0 and x is []int{3, 5, 3}
```

In assignments, each value must be assignable to the type of the operand to which it is assigned, with the following special cases:

1. Any typed value may be assigned to the blank identifier.
2. If an untyped constant is assigned to a variable of interface type or the blank identifier, the constant is first implicitly converted to its default type.
3. If an untyped boolean value is assigned to a variable of interface type or the blank identifier, it is first implicitly converted to type `bool`.

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt
| Block ) ] .
if x > max {
            x = max
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
            return x
} else if x > z {
            return z
} else {
            return y
}
```

Switch statements

"Switch" statements provide multi-way execution. An expression or type is compared to the "cases" inside the "switch" to determine which branch to execute.

```
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
```

There are two forms: expression switches and type switches. In an expression switch, the cases contain expressions that are compared against the value of the switch expression. In a type switch, the cases contain types that are compared against the type of a specially annotated switch expression. The switch expression is evaluated exactly once in a switch statement.

Expression switches

In an expression switch, the switch expression is evaluated and the case expressions, which need not be constants, are evaluated left-to-right and top-to-bottom; the first one that equals the switch expression triggers execution of the statements of the associated case; the other cases are skipped. If no case matches and there is a "default" case, its statements are executed. There can be at most one default case and it may appear anywhere in the "switch" statement. A missing switch expression is equivalent to the boolean value `true`.

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" {
ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
```

If the switch expression evaluates to an untyped constant, it is first implicitly converted to its default type. The predeclared untyped value `nil` cannot be used as a switch expression. The switch expression type must be comparable.

The switch expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
switch tag {
default: s3()
case 0, 1, 2, 3: s1()
case 4, 5, 6, 7: s2()
}

switch x := f(); {  // missing switch expression means "true"
case x < 0: return -x
default: return x
}

switch {
case x < y: f1()
case x < z: f2()
case x == 4: f3()
}
```

Implementation restriction: A compiler may disallow multiple case expressions evaluating to the same constant. For instance, the current compilers disallow duplicate integer, floating point, or string constants in case expressions.

Type switches

A type switch compares types rather than values. It is otherwise similar to an expression switch. It is marked by a special switch expression that has the form of a type assertion using the keyword `type` rather than an actual type:

```
switch x.(type) {
// cases
}
```

Cases then match actual types `T` against the dynamic type of the expression `x`. As with type assertions, `x` must be of interface type, but not a type parameter, and each non-interface type `T` listed in a case must implement the type of `x`. The types listed in the cases of a type switch must all be different.

```
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" {
TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type"
")" .
TypeCaseClause  = TypeSwitchCase ":" StatementList .
TypeSwitchCase  = "case" TypeList | "default" .
```

The TypeSwitchGuard may include a short variable declaration. When that form is used, the variable is declared at the end of the TypeSwitchCase in the implicit

block of each clause. In clauses with a case listing exactly one type, the variable has that type; otherwise, the variable has the type of the expression in the TypeSwitchGuard.

For statements

A "for" statement specifies repeated execution of a block. There are three forms: The iteration may be controlled by a single condition, a "for" clause, or a "range" clause.

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

For statements with single condition

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to the boolean value `true`.

```
for a < b {
            a *= 2
}
```

For statements with `for` clause

A "for" statement with a ForClause is also controlled by its condition, but additionally it may specify an *init* and a *post* statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not. Variables declared by the init statement are re-used in each iteration.

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
for i := 0; i < 10; i++ {
            f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to the boolean value `true`.

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

For statements with `range` clause

A "for" statement with a "range" clause iterates through all entries of an array, slice, string or map, or values received on a channel. For each entry it assigns *iteration values* to corresponding *iteration variables* if present and then executes the block.

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

The range expression `x` is evaluated once before beginning the loop, with one exception: if at most one iteration variable is present and `len(x)` is constant, the range expression is not evaluated.

Function calls on the left are evaluated once per iteration. For each iteration, iteration values are produced as follows if the respective iteration variables are present:

```
Range expression                          1st value          2nd value

array or slice  a  [n]E, *[n]E, or []E    index    i  int    a[i] E
string          s  string type            index    i  int    see below  rune
map             m  map[K]V                key      k  K      m[k] V
channel         c  chan E, <-chan E       element  e  E
```

Go statements

A "go" statement starts the execution of a function call as an independent concurrent thread of control, or *goroutine*, within the same address space.

```
GoStmt = "go" Expression .
```

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true }} (c)
```

Select statements

A "select" statement chooses which of a set of possible send or receive operations will proceed. It looks similar to a "switch" statement but with the cases all referring to communication operations.

```
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" StatementList .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
```

```
RecvExpr   = Expression .
```

A case with a RecvStmt may assign the result of a RecvExpr to one or two variables, which may be declared using a short variable declaration. The RecvExpr must be a (possibly parenthesized) receive operation. There can be at most one default case and it may appear anywhere in the list of cases.

Execution of a "select" statement proceeds in several steps:

1. For all the cases in the statement, the channel operands of receive operations and the channel and right-hand-side expressions of send statements are evaluated exactly once, in source order, upon entering the "select" statement. The result is a set of channels to receive from or send to, and the corresponding values to send. Any side effects in that evaluation will occur irrespective of which (if any) communication operation is selected to proceed. Expressions on the left-hand side of a RecvStmt with a short variable declaration or assignment are not yet evaluated.
2. If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection. Otherwise, if there is a default case, that case is chosen. If there is no default case, the "select" statement blocks until at least one of the communications can proceed.
3. Unless the selected case is the default case, the respective communication operation is executed.
4. If the selected case is a RecvStmt with a short variable declaration or an assignment, the left-hand side expressions are evaluated and the received value (or values) are assigned.
5. The statement list of the selected case is executed.

Since communication on `nil` channels can never proceed, a select with only `nil` channels and no default case blocks forever.

```
var a []int
var c, c1, c2, c3, c4 chan int
var i1, i2 int
select {
case i1 = <-c1:
            print("received ", i1, " from c1\n")
case c2 <- i2:
            print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
            if ok {
                        print("received ", i3, " from c3\n")
            } else {
                        print("c3 is closed\n")
            }
case a[f()] = <-c4:
            // same as:
            // case t := <-c4
            //          a[f()] = t
default:
            print("no communication\n")
}

for {  // send random sequence of bits to c
            select {
            case c <- 0:  // note: no statement, no fallthrough, no
folding of cases
            case c <- 1:
            }
}

select {}  // block forever
```

Return statements

A "return" statement in a function `F` terminates the execution of `F`, and optionally provides one or more result values. Any functions deferred by `F` are executed before `F` returns to its caller.

```
ReturnStmt = "return" [ ExpressionList ] .
```

In a function without a result type, a "return" statement must not specify any result values.

```
func noResult() {
            return
}
```

There are three ways to return values from a function with a result type:

The return value or values may be explicitly listed in the "return" statement. Each expression must be single-valued and assignable to the corresponding element of the function's result type.

```
1.  func simpleF() int {
2.     return 2
3.  }
4.
5.  func complexF1() (re float64, im float64) {
6.     return -7.0, -4.0
7.  }
```

The expression list in the "return" statement may be a single call to a multi-valued function. The effect is as if each value returned from that function were assigned to a temporary variable with the type of the respective value, followed by a "return" statement listing these variables, at which point the rules of the previous case apply.

```
8.   func complexF2() (re float64, im float64) {
9.      return complexF1()
10.  }
```

The expression list may be empty if the function's result type specifies names for its result parameters. The result parameters act as ordinary local variables and the function may assign values to them as necessary. The "return" statement returns the values of these variables.

```
11.  func complexF3() (re float64, im float64) {
12.     re = 7.0
13.     im = 4.0
14.     return
15.  }
16.
17.  func (devnull) Write(p []byte) (n int, _ error) {
18.     n = len(p)
19.     return
20.  }
```

Regardless of how they are declared, all the result values are initialized to the zero values for their type upon entry to the function. A "return" statement that specifies results sets the result parameters before any deferred functions are executed.

Implementation restriction: A compiler may disallow an empty expression list in a "return" statement if a different entity (constant, type, or variable) with the same name as a result parameter is in scope at the place of the return.

```
func f(n int) (res int, err error) {
            if _, err := f(n-1); err != nil {
                        return  // invalid return statement: err is
shadowed
            }
            return
}
```

Break statements

A "break" statement terminates execution of the innermost "for", "switch", or "select" statement within the same function.

```
BreakStmt = "break" [ Label ] .
```

If there is a label, it must be that of an enclosing "for", "switch", or "select" statement, and that is the one whose execution terminates.

```
OuterLoop:
            for i = 0; i < n; i++ {
                        for j = 0; j < m; j++ {
                                    switch a[i][j] {
                                    case nil:
                                                state = Error
                                                break OuterLoop
                                    case item:
                                                state = Found
                                                break OuterLoop
                                    }
                        }
            }
```

Continue statements

A "continue" statement begins the next iteration of the innermost enclosing "for" loop by advancing control to the end of the loop block. The "for" loop must be within the same function.

```
ContinueStmt = "continue" [ Label ] .
```

If there is a label, it must be that of an enclosing "for" statement, and that is the one whose execution advances.

```
RowLoop:
            for y, row := range rows {
                        for x, data := range row {
                                    if data == endOfRow {
                                                continue RowLoop
                                    }
                                    row[x] = data + bias(x, y)
                        }
            }
```

Goto statements

A "goto" statement transfers control to the statement with the corresponding label within the same function.

```
GotoStmt = "goto" Label .
goto Error
```

Executing the "goto" statement must not cause any variables to come into scope that were not already in scope at the point of the goto. For instance, this example:

```
            goto L  // BAD
            v := 3
L:
```

is erroneous because the jump to label L skips the creation of v.

A "goto" statement outside a block cannot jump to a label inside that block. For instance, this example:

```
if n%2 == 1 {
            goto L1
}
for n > 0 {
            f()
            n--
L1:
            f()
```

```
            n--
}
```

is erroneous because the label `L1` is inside the "for" statement's block but the `goto` is not.

Packages

Go programs are constructed by linking together *packages*. A package in turn is constructed from one or more source files that together declare constants, types, variables and functions belonging to the package and which are accessible in all files of the same package. Those elements may be exported and used in another package.

Source file organization

Each source file consists of a package clause defining the package to which it belongs, followed by a possibly empty set of import declarations that declare packages whose contents it wishes to use, followed by a possibly empty set of declarations of functions, types, variables, and constants.

```
SourceFile       = PackageClause ";" { ImportDecl ";" } {
TopLevelDecl ";" } .
```

Package clause

A package clause begins each source file and defines the package to which the file belongs.

```
PackageClause  = "package" PackageName .
PackageName    = identifier .
```

The PackageName must not be the blank identifier.

```
package math
```

A set of files sharing the same PackageName form the implementation of a package. An implementation may require that all source files for a package inhabit the same directory.

Import declarations

An import declaration states that the source file containing the declaration depends on functionality of the *imported* package (§Program initialization and execution) and enables access to exported identifiers of that package. The import names an identifier (PackageName) to be used for access and an ImportPath that specifies the package to be imported.

```
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" }
")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

The PackageName is used in qualified identifiers to access exported identifiers of the package within the importing source file. It is declared in the file block. If the PackageName is omitted, it defaults to the identifier specified in the package clause of the imported package. If an explicit period (.) appears instead of a name, all the package's exported identifiers declared in that package's package block will be declared in the importing source file's file block and must be accessed without a qualifier.

The interpretation of the ImportPath is implementation-dependent but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

Implementation restriction: A compiler may restrict ImportPaths to non-empty strings using only characters belonging to Unicode's L, M, N, P, and S general categories (the Graphic characters without spaces) and may also exclude the characters `!"#$%&'()*,:;<=>?[\]^`{|}` and the Unicode replacement character U+FFFD.

Consider a compiled a package containing the package clause `package math`, which exports function `Sin`, and installed the compiled package in the file identified by `"lib/math"`. This table illustrates how `Sin` is accessed in files that import the package after the various types of import declaration.

```
import declaration          Local name of Sin

import   "lib/math"         math.Sin
import m "lib/math"         m.Sin
import . "lib/math"         Sin
```

An import declaration declares a dependency relation between the importing and imported package. It is illegal for a package to import itself, directly or indirectly, or to directly import a package without referring to any of its exported identifiers. To import a package solely for its side-effects (initialization), use the blank identifier as explicit package name:

```
import _ "lib/math"
```

Program initialization

The packages of a complete program are initialized stepwise, one package at a time. If a package has imports, the imported packages are initialized before initializing the package itself. If multiple packages import a package, the imported package will be initialized only once. The importing of packages, by construction, guarantees that there can be no cyclic initialization dependencies. More precisely:

Given the list of all packages, sorted by import path, in each step the first uninitialized package in the list for which all imported packages (if any) are already initialized is initialized. This step is repeated until all packages are initialized.

Package initialization—variable initialization and the invocation of `init` functions—happens in a single goroutine, sequentially, one package at a time.
An `init` function may launch other goroutines, which can run concurrently with the initialization code. However, initialization always sequences the `init` functions: it will not invoke the next one until the previous one has returned.

Program execution

A complete program is created by linking a single, unimported package called the *main package* with all the packages it imports, transitively. The main package must have package name `main` and declare a function `main` that takes no arguments and returns no value.

```
func main() { … }
```

Program execution begins by initializing the program and then invoking the function `main` in package `main`. When that function invocation returns, the program exits. It does not wait for other (non-`main`) goroutines to complete.

Errors

The predeclared type `error` is defined as

```
type error interface {
            Error() string
}
```

It is the conventional interface for representing an error condition, with the nil value representing no error. For instance, a function to read data from a file might be defined:

```
func Read(f *File, b []byte) (n int, err error)
```

Run-time panics

Execution errors such as attempting to index an array out of bounds trigger a *run-time panic* equivalent to a call of the built-in function `panic` with a value of the implementation-defined interface type `runtime.Error`. That type satisfies the predeclared interface type `error`. The exact error values that represent distinct run-time error conditions are unspecified.

```
package runtime

type Error interface {
            error
            // and perhaps other methods
}
```

URL: https://go.dev/ref/spec

Lang 2

Sofia Specification

PART I - INFORMAL LANGUAGE SPECIFICATION

"Everything should be made as simple as possible, but not simpler"
Albert Einstein

The original specification for the language used in Compilers Course at Algonquin College was developed by **Prof. Svillen Ranev**.

- Originally, the language was called "PLATYPUS" - acronym for a ***P**rogramming **L**anguage – **A**forethought **T**iny **Y**et **P**rocedural, **U**nascetic and **S**ophisticated*.
- This version contains some modifications over the original language and some other elements created in **Sofia** (from Greek "Σοφια", or "*Sapientia*" in Latin), that means "wisdom".

The main purpose is to prove to create opportunity for C developers increase their knowledge and experience in this language by developing a **front-end compiler** in C. This document is based in the original **Sofia** language specification, shown as an introduction. Further, more concepts can be explored in the **BNF** and **Grammar** that formally define **Sofia**.

2.1. LANGUAGE OVERVIEW

The ***Sofia language*** is small and lacks some features like functions and complex data types, but in spite of that it is a **Turing complete** language.

Note 1: Turing Machine Languages

Defined by Allan Turing, the concept of these languages is that they can be operated by a simple control (automaton) that can read all the input from one tape and process according to a previous defined instructions (what we can call as a controller, origin of our nowadays CPUs).

A Turing complete language must be able to describe all the computations a Turing machine can perform, because it is proven that a Turing machine can perform (may be not very efficiently) any known computation described by an algorithm on a computer.

A programming language is Turing complete if it provides integer variables and standard arithmetic and

sequentially executes statements, which include assignment, simple selection and iteration statements.

A ***Sofia*** program is a single file program that has the following format:

```
# Main program #
main& {
      # Variables session #
      data {
            #
            int age$, id$;
            real salary%;
            string name@;
            #
      }
      code {
            # Optional statements session
            ... STATEMENTS ...
            Ex: input&, print&, etc.
            #
      }
}
# Optional function session
OPTIONAL_FUNCTION& {
}
#
```

- Differently from Python, indentation is not required: it is used only for didactics (and, obviously, make the code better understandable).
- Functions are followed by ampersand (&). Ex: main&.
  - These functions are classified as MNID (Method Name Identifier).
- The main program is called **main&** and it is composed by two sessions:
  - **data**: when you declare all variables to be used;
  - **code**: when we have the statements (that can be *optional*) and compose the **main function body**.
- Any number of statements (maybe no statements at all) can be included in the body of a program (that is, the body of a program could be empty).
- The language must allow for comments to be included in the program text.
  - The language supports multiple-line type comments.
  - A comment begins with one single # (at) and ends with another # allowing multiple lines.
- The language must be able to handle (program) computational tasks involving numbers (integer and float point numbers) or strings.
  - That is, the language can only handle whole (integer) numbers and numbers with a fractional part, both in constants and in variables.

  - The internal memory size of an **integer number** must be **2 bytes**.
  - Float point numbers must be represented internally using **4 bytes**.
  - Number type variables can hold **positive**, **zero**, or **negative** value numbers.
  - Number type literals (constants) can represent non-negative values only, that is, zero or positive values.
- The language must be able process inputs both as **variables** and as **constants**.
- Function names are ended by & (ampersand) and they are classified as **MNID** (Method Name Identifier).
- About the function sessions:
  - **data**: The **names of the variables** (*variable identifiers* = ***VID***) are composed of ASCII letters, digits or underscore (_).
    - A *variable identifier* (***VID***) can be of any length but only the first 8 characters are significant.
    - We have three kinds: **AVID** (arithmetic) that can be **IVID** (integer) or **FVID** (real) or string variables (**SVID**).
    - *Variable identifiers* are case sensitive.
    - By default, the type of a variable is defined by the first symbol of the variable identifier:
      - If a variable name ends with "**$**", the variable must be considered an Integer (**int**).
      - If it ends with "**%**", we are dealing with a float-point variable (**real**).
      - If it ends with "@", it should represent a sequence of chars (**string**).
      - **Note:** All other letters as initial character, are not considered to be a valid variable, but can be part of a **keyword** or a function name.
  - **code**: Operations supported in includes:
    - Classical **arithmetic operations** (both for integers and floating-point):
      - Addition (**+**), subtraction (**-**), multiplication (*****), division (***/***) or power (**^**).
    - One single **string operation**: concatenation (represented by **++** symbol) operation must be allowed for both string variables and string constants.
    - Expressions (both arithmetic and strings).
- An **integer literal** (***IL***) or **integer constant** is a non-empty string of ASCII digits.
  - Only *decimal integer literals* (integer numbers written in base 10) are supported.
  - A *decimal* **constant** representation is any finite sequence (string) of ASCII digits that, if it has a length more than one digit, should contain digits.
    - Examples of legal decimal constant representations are 0, 00, 013, 171, 1024.

    - Note that 32768 is a legal decimal number representation, but it is illegal as a value for integer literals – (out of range).
- A *floating-point literal* (***FL***) consists of digits followed by a **period** (**.**) and eventual digits after it.
    - The first string (not-null sequence of digits), the dot and the second string (another sequence of digits) must be always present. The first string should always be a legal decimal literal.
    - Examples of floating-point constant representations are 003.0, 7203.000, 13.45.
    - Examples of non-acceptable floating-points are 42., .325.
- A *string literal* (**SL**) is a finite sequence of ASCII characters enclosed in single quotes (‘).
    - The string literal cannot contain ‘ as part of the string literal.
    - **Empty string** is allowed, (that is, ‘’ is a legal string literal).
- There is explicit data type declaration in the language:
    - All variables must be declared in an initial session of data containing:
        - **int** varname11$, varname12$, …
        - **real** varname21%, varname22%, …
        - **string** varname31@, varname32@, …
- The language should allow **mixed type** arithmetic expressions and arithmetic assignments.
    - Mixed arithmetic expressions are always evaluated as floating point.
    - **Automatic conversions** (promotions and demotions) must take place in such cases.

2.2. LANGUAGE STATEMENTS

The language must provide the following statement types:

2.2.1. Block statements:

- All functions (including **main&**) must have two sections: **data** and **code**:
    - The ***data*** must declare all variables according to their type (each variable must be declared in the appropriate line):

```
data {
    int #list of integer variables#
    real #list of floating-point variables#
    string #list of string variables#
}
```

        - **NOTE:** If there are no variables of one specific type, this datatype should not appear.

- The ***code*** represents the section that contains all statements (assignment, selection, iteration, input or output):

```
code {
    #optional statements#;
}
```

  - **NOTE:** If there are no statements, a single line ending with semi-colon (;) will be found.

2.2.2. Assignment Statement:

- ***Assignment Statement*** is an ***Assignment Expression*** terminated with an *End-Of-Statement (EOS)* symbol (semicolon **;**).
  - The *Assignment Expression* has one of the following two forms:

```
AVID = Arithmetic Expression
SVID = String Expression
```

where ***AVID*** is Arithmetic *VID* and ***SVID*** is String *VID*.

  - The *Arithmetic Expression* is an infix expression constructed from arithmetic variable identifiers, constants, and arithmetic operators. The following operators are defined on arithmetic variables and constants:

```
Addition: +
Subtraction: –
Multiplication: *
Division: /
Power: ^
```

  - The operators follow the **standard associativity** and **order of precedence** of the arithmetic operations.
    - Parentheses **( )** are also allowed. '**+**' and '**–**' may be used as unary sign operators only in single-variable, single-constant, or single **()** expressions.
      - Examples: –a#, +5.0, –3, and –(b#–5.0) are legal expressions.
      - Examples: – – a#, and a% +– b% are illegal expressions;
    - The **default sign** (no sign) of the arithmetic literals and variables is positive (+).
    - The *Arithmetic Expression* evaluates to a numeric value (integer or real number).
      - Examples:

```
sum% = 0.0;
first% = -1.0;
```

```
second% = -(1.2 + 3.0);
third$ = 7; fourth$ = 04;
sum% = first% + second%  - (third$ + fourth$);
```

- The *String Expression* is an **infix expression** constructed from string variable identifiers, string constants, and string operators.
  - Only one string manipulation operator is defined on string variables and constants:

string *concatenation* or catenation (or append): **++**

  - The **++** is a binary operator and the order of evaluation (**associativity**) of the concatenation operator is from left to right. Parentheses **( )** are not allowed. The *String Expression* evaluates to a string.
  - Examples:

```
light@ = 'sun';
day@ = 'Let the ' ++ light@ ++ 'shines!'; # 'Let the sun shines!' #
```

2.2.3. Selection Statement:

```
if pre-condition (Conditional Expression)
then {
      statements (optional - may contain no statements at all)
}
else {
      statements (optional - may contain no statements at all)
};
```

- Both **THAN** and **ELSE** clauses must be present but may be empty – no statements at all.

2.2.4. Iteration Statement:

```
while (Conditional Expression) do {
      statements
};
```

- The *Iteration Statement* executes repeatedly the statements specified by the **DO** clause of the **WHILE** loop depending on the *pre-condition* and *Conditional Expression*.
- *Conditional Expression* is a left-associative infix expression constructed from relational expression(s) and the logical operators:
  - **.and.** (logical binary operator AND),
  - **.or.** (logical binary operator OR) and

- **.not.** (logical unary operator NOT).

- The operator **.not.** has higher order of precedence than the others and **.and.** has higher order of precedence than **.or.**.
- The evaluation of the conditional expression stops abruptly if the result of the evaluation can be determined without further evaluation (short-circuit evaluation).
- The *Conditional Expression* evaluates to true or false. The values of true and false are implementation dependent.
- *Relational Expression* is a binary (two operands) infix expression constructed from variable identifiers and constants of compatible types and comparison operators **==**, **<>**, **<**, **>**.
  - The comparison operators have a higher order of precedence than the logical operators.
  - The *Relational Expression* evaluates to true or false. The values of true and false are implementation dependent.
  - Example:

```
while (balance% >0.0 .and. answer@=='y') # Additive test #
do {
   balance% = balance% - debit%;
}
```

2.2.5. Input/Output Statements:

- The **input&** statement receives values from the standard input and assigns them to the variables in the *variable list* in the same order they are listed in the *variable list*.
  - *variable list* is a list of one or more *VID*s separated with commas.

```
input& (variable list);
```

- Example:

```
input&(a$, b$, c$);
```

- The **print&** statement sends the string literal or the values of the variable in the *variable list* to the standard output.
  - print&() with no parameter outputs a line terminator.

```
print& (string literal);
```

- Example:

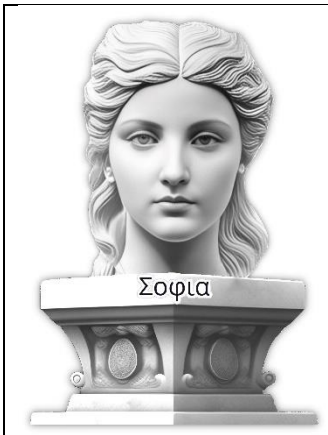
```
print& ('Sofia is a strange language');
print& (Name@);
print& ();
```

2.3. KEYWORDS AND SOME OTHER TOKENS

- The words (lexemes) **data, code, int, real, string, if, then, else, while, do** are **keywords** and they cannot be used as variable identifiers.
  - **Note:** In case of functions (ex: main&, input&, print&) are considered as internal functions (should not be overridden).
- Finally, about the tokens in **Sofia**:
  - Blanks (spaces), horizontal and vertical tabs, new lines, form feeds, and comments are ignored.
  - If they separate tokens, they are ignored after the token on the left is recognized. Tokens (except for string literals), that is, variable identifiers, integer literals, floating-point literals, keywords and two-character operators may not extend across line boundaries.

PART II - FORMAL LANGUAGE SPECIFICATION

"Grammar, which knows how to control even kings"
Molière, Les Femmes Savantes (1672), Act II, scene vi



This section continues the presentation of **Sofia**[7] language (adapted from original "PLATYPUS" language created by **Prof. Svillen Ranev**) and will present its **grammar** using a **BNF** (Backus-Naur Form) representation.

- BNF is a common way to represent all structures from your language.
- In other words, a context-free grammar is used to define the lexical and syntactical parts of the **Sofia** language and the lexical and syntactic structure of a program

2.4. GRAMMAR INTRODUCTION

Note 1: Context-Free Grammars

*A **context-free grammar (CFG)**, often called **Backus Normal Form** or Backus-Naur Form **(BNF) grammar**, consists of four finite sets: a finite set of terminals; a finite set of non-terminals; a finite set of productions; and a start or a goal symbol.*

*One of the sets consists of a **finite number of productions** (also called **replacement rules**, **substitution rules**, or **derivation rules**).*

[7] **Sofia** is a fake language that should be adaptated to your programming language.

*Each production has an abstract symbol called a **nonterminal** as its left-hand side, and a sequence of **one or more nonterminal and terminal symbols** as its right-hand side.*

*For each grammar, the terminal symbols are drawn from a specified **alphabet**.*

*Starting from a sentence consisting of a single distinguished nonterminal, called the **start symbol**, a given context-free grammar specifies a language, namely, the **infinite set of possible sequences** of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.*

2.4.1. The Sofia Lexical Grammar

- A ***lexical grammar*** for **Sofia** is given as follows.
  - This grammar has as its terminal symbols the characters of the ASCII character set.
  - It defines a set of productions, starting from the start symbol ***input***, that describe how sequences of ASCII characters are translated into a sequence of input elements.
  - These input elements, with white space and comments discarded, form the terminal symbols for the syntactic grammar for Sofia and are called ***tokens***.
  - These tokens are the variable identifier, keyword, integer literal, floating-point literal, string literal, separator, and operator of the Sofia language.

2.4.2. The Sofia Syntactic Grammar

- The incomplete ***syntactic grammar*** for **Sofia** is given as follows.
  - This grammar has tokens defined by the lexical grammar as its terminal symbols.
  - It is to define a set of productions, starting from the start symbol <program> that describe how sequences of tokens can form syntactically correct programs.

2.4.3. Grammar Notation

- Terminal symbols are shown in normal font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol.
  - These are to appear in a program exactly as written.
- Non-terminal symbols are shown in **triangular brackets** <nonterminal> for ease of recognition.
  - However, **non-terminals** can also be recognized by the fact that they appear on the left-hand sides of productions.
  - The definition of a non-terminal is introduced by the **name** of the nonterminal being defined followed by an **arrow** (→) sign.

- One or more alternative right-hand sides for the nonterminal then follow on succeeding line(s) preceded by the **alternation** symbol (|).
- The symbol will represent the empty or null string. Thus, a production

A → ε

states that A can be replaced by the empty string, effectively **erasing** it.

- When the words "***one of*** " follow the ε in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition.
- For example, the production:

<small letters from a to c> → one of {a, b, c]

is not a standard BNF operation but is merely a convenient abbreviation for:

<small letters from a to c> → a | b | c

- The **right-hand side** of a lexical production may specify that certain expansions are not permitted by using the phrase "*but not*" and then indicating the expansions to be excluded, as in the productions for <input character>

<input character> → one of {ASCII characters but not SEOF}

- The prefix **opt**_, which may appear before a terminal or nonterminal, indicates an ***optional symbol or element***.

2.4.4. Lexical Specification

This section specifies the **lexical grammar** (structure) of **Sofia**.

- **Sofia** programs are **single file programs** written in ASCII.
  - Lines are terminated by the ASCII characters **CR**, or **LF**, or by the combination CR LF.
  - Source files are terminated by the SEOF character. This character is **Control-Z** in DOS or **Control-D** in UNIX.
  - The ASCII characters are reduced to a sequence of input elements, which are white space, comments, and tokens.
  - The tokens are the variable identifier, keyword, integer literal, floating-point literal (real), string literal, separator and/or operator of the Sofia syntactic grammar.

2.4.4.1. Input Elements and Tokens

- The input characters and line terminators that result from input line recognition are reduced to a sequence of <input elements>.

- Those input elements that are not white space or comments are ***tokens***. The tokens are the terminal symbols of the Sofia syntactic grammar.
- This process is specified by the following productions:

<input character> → one of {ASCII characters but not SEOF}

<input> → <input elements> SEOF

<input elements> → <input element>

| <input elements> <input element>

<input element> → <white space>

| <comment>

| <token>

<token> → <variable identifier>

| <keyword>

| <floating-point literal>

| <integer literal>

| <string literal>

| <separator>

| <operator>

2.4.4.2. White Space

- White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.

<white space> → the ASCII SP character, also known as “space”

| the ASCII HT character, also known as “horizontal tab”

| the ASCII VT character, also known as “vertical tab”

| the ASCII FF character, also known as “form feed”

| <line terminator>

<line terminator> → CR | LF | CRLF

2.4.4.3. Separators

- The following eight ASCII characters are the Sofia separators (punctuators):

<separator> → one of { (,), {, }, ,, ; }

They can only be used in a specific context defined by the grammar**.**

2.4.5. Part III – Sofia Syntactic Specification

This section specifies the **grammar** of Sofia.

2.4.5.1. Main Program

The program is composed by one special function (whose name is supposed a kind of **MNID – Method Name Identifier**): "main&" that is similar to **C** / **Java** and it is defined as follows.

```
<program> → main& {
                <data_session>
                <code_session>
                }
```

- The first part (**data**) is the place we declare the variables:

```
<data_session> → data {
                <opt_varlist_declarations>
}
```

  - Here are the definitions for each variable type:

```
<opt_varlist_declarations> → <varlist_declarations> | ϵ
```

  - All variable declarations are defined as follows:

```
<varlist_declarations> → <varlist_declaration>
                       | <varlist_declarations><varlist_declaration>
```

  - The list of variables can be defined:

```
<varlist_declaration> → <integer_varlist_declaration>
                      | <float_varlist_declaration>
                      | <string_varlist_declaration>
```

  - And we have proper ways to define variables according to their type:

```
<integer_varlist_declaration>  →  int <integer_variable_list>;
<float_varlist_declaration>    →  real <float_variable_list>;
<string_varlist_declaration>   →  string <string_variable_list>;
```

- The sequence of lists is given here:

```
<integer_variable_list>   →   <integer_variable_identifier>
                              | <integer_variable_list>, <integer_variable_identifier>
<float_variable_list>     →   <float_variable_identifier>
                              | <float_variable_list>, <float_variable_identifier>
<string_variable_list>    →   <string_variable_identifier>
                              | <string_variable_list>, <string_variable_identifier>
```

- These variables will be declared later.

- The second part (**code**) is in fact where we can find optional statements (<opt_statements>)

```
<code_session> → code {
                     <opt_statements>
                     }
```

is not a standard BNF operation but is merely a convenient abbreviation for:

```
<opt_statements> → { <statements> } | { }
```

which in turn is abbreviation for:

```
<opt_statements> → <statements> | ε
```

The non-terminal statements will be declared later.

2.4.5.2. Comments

- Sofia supports **multi-line comments**: a comment is a text beginning and ending with #.
- A comment is formally specified by the following grammar productions:

```
<comment> → # <opt_characters> #
<opt_characters> → <opt_characters> <input character> | ε
```

2.4.5.3. Keywords

- The following character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers:

```
<keyword> →  data | code | int | real | string | if | then | else | while | do
```

2.4.5.4. Method Identifiers

- A method name identifier (***MNID***) is the way to define functions. A MID is a sequence of ASCII letters, numbers or underscore, ending with & (ampersand).
- Methods are supposed to have **data** and **code** sessions.
- The main method is main&.

2.4.5.5. Variable Identifiers

- A variable identifier (***VID***) can be arithmetic or textual (string). A VID is a sequence of ASCII letters, ASCII digits or underscore, whose initial symbol can vary according to the datatype:
  - In the case of integer (**IVID**), the variable must end with dollar - "**$**".
  - For **FVID** (float-pointing variables), the last char is a percentage (**%**), followed by the letters, numbers and underscore.
  - In the case of strings (**SVID**), you must have letters, numbers and underscore ending with at symbol (@).
- A *VID* can be of any length but only the first 8.
- A variable is a storage location and has an associated data type.
  - The **Sofia** language supports only three data type: integer, real and string data type.
  - Determining the type of the **arithmetic** variable (integer or the floating-point) is built in the grammar, obeying the syntax definitions.

<variable identifier> → <arithmetic_variable_identifier> |
<string_variable_identifier>

<arithmetic_variable identifier> → <integer_variable_identifier> | <float_variable_identifier>

<integer_variable_identifier> → <letters or digits or underscore>#

<float_variable_identifier> → <letters or digits or underscore>%

<string_variable_identifier> → <letters or digits or underscore>$

<letters or digits or underscore> → <letter or digit or underscore>
| <letters or digits or underscore> <letter or digit or underscore>

<letter or digit or underscore> → one of {a…z, A...Z, 0…9, _}

- Note that each VID must have at least one single letter.

2.4.5.6. Method Identifiers

- A method identifier (***MID***) is a sequence of ASCII letters, ASCII digits or underscore, whose last symbol is an ampersand (&).
    - For instance, the most important MID is the main: "**main&**".

2.4.5.7. Integer Literals

- An ***integer literal*** (constant) is the source code representation of an integer decimal value or integer number.
- The **Sofia** language supports two deferent representations of integer literal: zero decimal integer literal and non-zero decimal integer literal.
- The **internal (machine) size** of an integer number must be 2 bytes.
- The literals by default are non-negative, but their sign can be changed at run-time by applying unary sign arithmetic operation.

<integer literal> → <digits>

<digits> → <digit> | <digits> <digit>

<digit> → one of {0 1 2 3 4 5 6 7 8 9}

2.4.5.8. Floating-point Literals

- A ***floating-point literal*** is the source code representation of a fixed decimal value.
- The numbers must be represented internally as floating-point numbers.
- The **internal size** must be 4 bytes.
- The **literals** by default are non-negative, but their sign can be changed at run-time by applying unary sign arithmetic operation.

<floating-point literal> → <digits> . <digits>

2.4.5.9. String Literals

- A ***string literal*** is a sequence of ASCII characters enclosed in single quotation marks.
- The ‘ and source-end-of-file character SEOF cannot be a string character.
- **Note**: SEOF is implementation dependent.

<string literal> → "<opt_string characters>"

<opt_string characters> → <string characters> | ε

<string characters> → <string character>

| <string characters> <string character>

<string character> → one of {ASCII characters but not SEOF}

2.4.5.10. Operators

- The following tokens are the **Sofia** operators, formed from ASCII characters:

<operator> → <arithmetic operator> | <string concatenation operator>

| <relational operator> | <logical operator>

| <assignment operator>

<arithmetic operator> → one of {+, -, *, /, ^ }

<string concatenation operator> → ++

<relational operator> → one of { <, >, ==, <> }

<logical operator> → **.and.** | **.or.** | **.not.**

<assignment operator> → =

2.4.5.11. Statements

- The sequence of execution of a **Sofia** program is controlled by statements.
  - Some statements contain other statements as part of their structure; such other statements are sub statements of the statement.

<statements> → <statement> | <statements> <statement>

- The definition of statement is given as follows.
  - **Sofia** supports the following **five types** of statements: assignment, selection, iteration, input and output statements.

<statement> → <assignment statement>

| <selection statement>

| <iteration statement>

| <input statement>

| <output statement>

A. Assignment Statement

- The assignment statement is evaluated in the following order.

- First, the assignment expression on the right side of the assignment operator is evaluated.
- Second, the result from the evaluation is stored into the variable on the left side of the assignment operator.
- If the assignment expression is of arithmetic type and the data types of the variable and the result are different, the result is converted to the variable type implicitly.
  - Arithmetic variable identifiers (**AVID**) have restriction about the symbols (ending with one specific symbol - **$** for integers and **%** for float-point literals / reals) followed by letters, digits or underscore).
  - String expressions operate on strings only and no conversions are allowed. Remember that string variables identifier (**SVID**) follows similar pattern for names and should finish with at symbol (@).

<assignment statement> → <assignment expression>;

<assignment expression> → <arithmetic_variable identifier> = <arithmetic expression>
| <string_variable identifier> = <string expression>

B. Selection Statement (if statement)

- The ***selection statement*** is an alternative selection statement, that is, there are two possible selections.
  - If the *conditional expression* evaluates to TRUE, the statements (if any) contained in the **then** clause are executed, and the execution of the program continues with the statement following the selection statement.
  - If the *conditional expression* evaluates to FALSE, only the statement (if any) contained in the **else** clause are executed and the execution of the program continues with the statement following the selection statement.
- Both **then** and **else** clauses must be present but may be empty – no statements at all.

<selection statement> → **if** (<conditional expression>)
then { <opt_statements> }
else { <opt_statements> } ;

C. Iteration Statement (the loop statement)

- The ***iteration statement*** is used to implement iteration control structures.
- The ***iteration statement*** executes repeatedly the statements specified by the **do** clause of the **while** loop depending on the *pre-condition* and *conditional expression*.
- If the **conditional expression** is true, the statements are repeated until the evaluation of the *conditional expression* becomes false.

- Otherwise, the program continues after the iteration statement.

<iteration statement> → **while** (<conditional expression>)
do { <statements>};

D. Input Statement

- The ***input statement*** reads a floating-point, an integer or a string literal from the standard input and stores it into a floating-point, an integer variable or a string variable.

<input statement> → **input&** (<variable list>);
<variable list> → <variable identifier> | <variable list>,<variable identifier>

- Remember:

<variable identifier> → <integer_variable> | <float_variable> | <string_variable>

E. Output Statement

- The ***output statement*** writes a variable list or a string to the standard output.
- Output statement with an empty variable list prints an empty line.

<output statement> → **print&** (<opt_variable list>);
| **print&** (<string literal>);

2.4.5.12. Expressions

This section specifies the meanings of **Sofia** expressions and the rules for their evaluation.

- Most of the work in a **Sofia** program is done by evaluating expressions, either for their side effects, such as assignments to variables, or for their values, which can be used as operands in larger expressions, or to affect the execution sequence in statements, or both.
- An expression is a sequence of operators and operands that specifies a computation.
- When an expression in a **Sofia** program is *evaluated* (*executed*), the result denotes a value.
- There are four of expressions in the **Sofia** language: arithmetic expression, string expressions, relational expressions, and conditional expression.
- The expressions are always evaluated from left to right.

A. Arithmetic Expression

- An ***arithmetic expression*** (using **IVID**, **FVID**, **IL** or **FPL**) is an infix expression constructed from arithmetic variables, arithmetic literals, and the operators *plus* (+), *minus* (-), *multiplication* (*), *division* (/) and *power* (^).

- The arithmetic expression always evaluates either to a floating-point value or to an integer value.
- Mixed type arithmetic expressions and mixed arithmetic assignments are allowed.
- The data type of the result of the evaluation is determined by the data types of the operands.
- If there is at least one floating-point operand, all operands are converted to floating-point type, the operations are performed as floating-point, and the type of the result is floating-point.
- The type conversion (**coercion**) is implicit.
- All operators are left associative.
- Plus and minus can be used as unary operator to change the sign of a value. These operators have the same order of precedence.
- Multiplication and division have the same order of precedence, but they have a higher precedence than plus and minus operators. This time, power is included as extra operator.
- In this case they have the highest order of precedence, and they are evaluated first.

The **formal syntax** of the arithmetic expression is listed below.

<arithmetic expression> → <unary arithmetic expression>
| <additive arithmetic expression>

<unary arithmetic expression> → <primary arithmetic expression>
| + <primary arithmetic expression>

<additive arithmetic expression> →
<additive arithmetic expression> + <multiplicative arithmetic expression>
| <additive arithmetic expression> - <multiplicative arithmetic expression>
| <multiplicative arithmetic expression>

<multiplicative arithmetic expression> →
<multiplicative arithmetic expression> * <primary arithmetic expression>
| <multiplicative arithmetic expression> / <primary arithmetic expression>
| <primary arithmetic expression>

<primary arithmetic expression> → <arithmetic variable identifier>
| <integer literal>
| <floating-point literal>
| (<arithmetic expression>)

B. String Expression

- A ***string expression*** is an infix expression constructed from string variables, string literals (**SL**), and the operator *append* or *concatenation* (**++**).
- The string expression always evaluates to a string (or a pointer to string).
- The append operator is left associative.

<string expression> → <primary string expression>

| <string expression> **++** <primary string expression>

<primary string expression> → <string variable identifier>

| <string literal>

C. Conditional Expression

- A ***conditional expression*** is an infix expression constructed from relational expressions and the logical operators **.not.**, **.and.** and/or **.or.**.
- Parentheses are not allowed in the conditional expressions; thus, the evaluation order cannot be changed. All operators are left associative.
- The conditional expressions evaluate to TRUE or FALSE.
- The **internal representation** of the values of true and false are left to the implementation.
- The formal syntax of the conditional expression follows.

<conditional expression> → <logical OR expression>

<logical OR expression> → <logical AND expression>

| <logical OR expression> **.or.** <logical AND expression>

<logical AND expression> → <logical NOT expression>

| <logical AND expression> **.and.** <logical NOT expression>

<logical NOT expression> → **.not.** <relational expression>

| <relational expression>

D. Relational Expression

- A ***relational expression*** is an infix expression constructed from variable identifiers (VID), literals (constants), and comparison operators (**==**, **<>**, **<**, **>**). The comparison operators have a higher order of precedence than the logical operators do.
- The relational expressions evaluate to true or false.
- The formal syntax of the relational expression follows.

<relational expression> → <relational a_expression>
| <relational s_expression>

<relational a_expression> →
<primary a_relational expression> **==** <primary a_relational expression>
| <primary a_relational expression> **<>** <primary a_relational expression>
| <primary a_relational expression> **>** <primary a_relational expression>
| <primary a_relational expression> **<** <primary a_relational expression>

<relational s_expression> →
<primary s_relational expression> **==** <primary s_relational expression>
| <primary s_relational expression> **<>** <primary s_relational expression>
| <primary s_relational expression> **>** <primary s_relational expression>
| <primary s_relational expression> **<** <primary s_relational expression>

<primary a_relational expression> → <floating-point literal>
| <integer literal>
| <arithmetic variable identifier>

<primary s_relational expression> → <primary string expression>

PART III – BASIC MODELS FOR SOFIA

The idea about creating basically a **Sofia** program to recognize Hello is based on the idea that you can detect the following code...

```
main& {
      data { }
      code {
            print&('Hello World!');
      }
}
```

Note that in **Sofia**, we can recognize basically the following elements:

- Methods (ending with "&"): **main&** and **print&**.
- Keywords (**data** and **code**).
- String literal ('Hello World!');

- Basic tokens: '{', '}', '(', ')', ';'.

3.1. REGULAR EXPRESSIONS

Using the Kleene Theorem, several models for any language can be applied. For example, RE (Regular Expression), TD (Transition Diagram = Automata) and TT (Transition Table).

- To use RE, we need to define lexeme classes. Ex:

  (0) L = [*A-Za-z*] (Letters)

  (1) D = [*0-9*] (Digits)

  (2) U = _ (Underscore)

  (3) M = *&* (Method delimitator)

  (4) Q = ' (String delimitator – single quotes)

  (5) O = [^**LDUMQ**] (Other chars)

- So, we can define RE for generic classes using the following definitions:

  MVID = **L(L|D|U)*M**

  SL = **Q[^Q]*Q** = **Q[L|D|U|M|O]*Q**

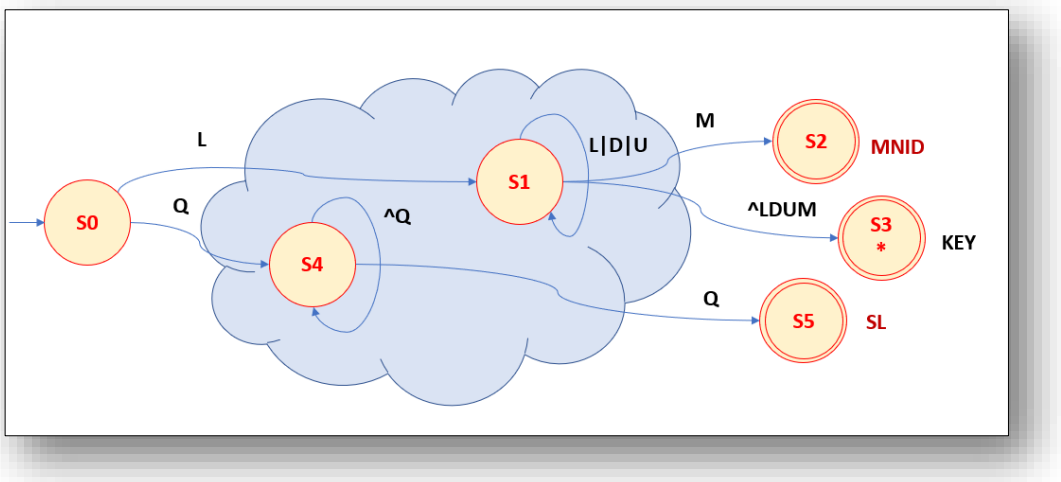
  KEY = { **data**, **code** }

3.2. AUTOMATA

Some tokens can be directly recognized (see the basic tokens above).

However, to detect other classes (ex: keywords, methods or literals), we need to use a generic approach to recognize Regular Expressions (**RE**).

So, it is possible to create an automaton to implement the RE, as indicated below.



3.3. TRANSITION TABLE

The automata implementation can be given by **TT** (Transition Table). The following example is indicating the way to recognize the minimal classes for the Hello World program:

- **MNID** (Method Name Identifier);
- **KEY** (Keywords)
- **SL** (String Literals).

Input	Input Symbol						Output
State \ Classes	0	1	2	3	4	5	Type
	L(A-Z)	D(0-9)	U(_)	M(&)	Q(')	O	-
S0	1	ESNR	ESNR	ESNR	4	ESNR	NOAS [0]
S1	1	1	1	2	3	3	NOAS [1]
S2	FS	FS	FS	FS	FS	FS	ASNR (MNID) [2]
S3	FS	FS	FS	FS	FS	FS	ASWR (KEY) [3]
S4	4	4	4	4	5	4	NOAS [4]
S5	FS	FS	FS	FS	FS	FS	ASNR (SL) [5]

Where:

- ESNR – Error State and No Retract;
- ESWR – Error State With Retract;
- FS – Final State.

And:

- ASNR – Accepting State and No Retract;
- ASWR – Accepting State With Retract;
- NOAS – Non-acceptable State.

Finally, it is possible to define the code to be used in the transition table:

```
static sofia_intg transitionTableMin[][TABLE_COLUMNS] = {
      /*          [A-z], [0-9],     _,     &,     ', other                  */
      /*           L(0),  D(1), U(2), M(3), Q(4), O(5)                     */
      /* S00 */ {     1, ESNR, ESNR, ESNR,    4, ESNR}, /* NOAS           */
      /* S01 */ {     1,    1,    1,    2,    3,    3}, /* NOAS           */
      /* S02 */ {    FS,   FS,   FS,   FS,   FS,   FS}, /* ASNR (MNID)    */
      /* S03 */ {    FS,   FS,   FS,   FS,   FS,   FS}, /* ASWR (KEY)     */
      /* S04 */ {     4,    4,    4,    4,    5,    4}, /* NOAS           */
      /* S05 */ {    FS,   FS,   FS,   FS,   FS,   FS}  /* ASNR (SL)      */
};
```

The final states represented by MNID (Method Names IDs) KEY (Keywords) and SL (String literal) must be recognized by specific functions in the code to be implemented in the **Scanner**.

- ***NOTE:*** *This model **is not complete** for Sofia language. The objective is only to provide a minimum model that can accept the "Hello World" program written in this hypothetical language.*

Sofia - Bulgary (source: Wikipedia).

Lang 3

More Languages Examples

3.1. Basic Examples

EXAMPLES 1 – SOFIA LANGUAGE

Ex.1.1. Hello World

```
# Sofia Hello #
  main& {
     data { }
     code {
            print&('Hello World!');
      }
}
```

Ex.1.2. Sphere Volume

```
# Sofia Example (Volume of a sphere) #
main& {
       data {
              real PI%, r%, Vol%;
       }
       code {
              PI% = 3.14;
              input&(r%);
              Vol% = 4.0 / 3.0 * PI% * (r% * r% * r%);
              print&(Vol%);
       }
}
```

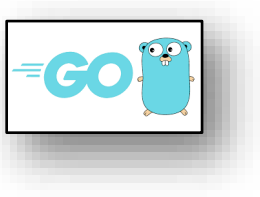
Ex.1.3. Data types

```
# Sofia Example
The program is lexically correct about the SOFIA city #
main& {
      data {
            int FoundationYear$;
            real Area%, HDI%;
            string CountryName@;
      }
      code {
            FoundationYear$ = -7000;
            Area% = 492.0;
            HDI% = 0.871;
            CountryName@ = 'Bulgary';
            print&(CountryName$);
      }
}
```

Arbitrary Program

```
# Sofia Example
  The program is "lexically" correct #
  main& {
      data {
            int i$;
            real a%, sum008%;
            string text@;
      }
      code {
            a%=+1.2;
            sum008% = 7.87050 ;
            print&(a%, sum008%);
            i$=0;
            while (i$ < 32767 .or. i$ == 32767)
            do {
                  i$ = i$ + 2;
                  a% = a%*i$/0.5 ;
                  sum008% = sum008% + a% - 1 ;
            };
            if (text@ == '')
            then {
                  text@ = 'prog' ++ 'ram';
            } else {
                  text@ = text@ ++ 'ram';
            };
            print&('\* This is a program -:)-<-<-- \*');
            print&(text@);
            if (text@ <> 'program' .and. sum008%==8.0 .or. i$>10)
            then {
                  print&(sum008%);
                  print&();
            }
            else{};
       }
}
```

EXAMPLES 2 – GO LANGUAGE



DEVELOPMENT NOTES:

These examples can run in your machine[8] or by using online resources[9].

Ex.2.1. Hello World

```
/* Go Hello World */
package main
import ("fmt")
func main() {
  fmt.Println("Hello World!")
}
```

Output:

```
Hello World!
```

Ex.2.2. Sphere Volume

```
/* Sphere Volume */
package main
import (
      "fmt"
      "math"
)
func sphereVolume(radius float64) float64 {
      return (4.0 / 3.0) * math.Pi * math.Pow(radius, 3)
}
func main() {
      radius := 5.0 // Replace this with the desired radius
      volume := sphereVolume(radius)
      fmt.Println("Volume of the sphere with radius",
                  radius, " is ", volume)
}
```

Output:

```
Volume of the sphere with radius 5 is 523.598775598299
```

[8] To run locally:

- Step 1: Download and install "Go" (https://go.dev/)
- Step 2: Adjust local variables (ex: GOPATH) and include the "%GOPATH%/bin" folder in your %PATH%.
- Step 3: Build the code (`go build <code>.go`) and/or run it (`go build <code>.go`)

[9] Online compilers:

[1] Tour of Go: https://go.dev/tour/welcome/1

[2] W3Schools: https://www.w3schools.com/go/trygo.php?filename=demo_helloworld

Ex.2.3. Data types

```
package main
import (
	"fmt"
	"reflect"
)
func main() {
	var countryName = "Bulgary"
	var foundationYear = -7000
	var area = 492.0
	fmt.Println("countryName datatype: ", reflect.TypeOf(countryName))
	fmt.Println("foundationYear datatype: ", reflect.TypeOf(foundationYear))
	fmt.Println("area datatype: ", reflect.TypeOf(area))
	fmt.Println(countryName, " was founded in: ", foundationYear, " and has ",
		area, " km2 area.")
}
```

Output:

```
countryName datatype: string
foundationYear datatype: int
area datatype: float64
Bulgary  was founded in: -7000 and has 492 km2 area.
```

Ex.2.4. Boolean Functions

```
package main

import "fmt"

type func_ptr func(int, int) int

/* Functions for Calls ................. */

func call_func(op func_ptr, a, b int) int {
	return op(a, b)
}

func call_default(f func_ptr) int {
    return call_func(f, 1, 0);
}

/* Functions for Constants …............ */

func TRUE(x int, y int) int {
	return x
}

func FALSE(x int, y int) int {
	return y
}

/* Unary functions ..................... */

func NOT(x func_ptr) int {
    return call_func(x, 0, 1);
}
```

```
/* Binary functions ..................... */

func AND(x func_ptr, y func_ptr) int {
    return call_func(x,
        call_default(y),
        call_default(FALSE));
}

func OR(x func_ptr, y func_ptr) int {
    return call_func(x,
        call_default(TRUE),
        call_default(y));
}

func XOR(x func_ptr, y func_ptr) int {
    return call_func(x,
        call_func(y,
            call_default(FALSE),
            call_default(TRUE)),
        call_default(y));
}

func IMP(x func_ptr, y func_ptr) int {
    return call_func(x,
        call_default(y),
        call_default(TRUE));
}

/* Main function ..................... */

func main() {
    t, f := TRUE, FALSE;
    T, F, n, a, o, x, i := 0, 0, 0, 0, 0, 0, 0;
    fmt.Println("Constants ................");
    t = TRUE;
    T = call_default(t);
    fmt.Println("TRUE: ", T);
    f = FALSE;
    F = call_default(f);
    fmt.Println("FALSE: ", F);
    fmt.Println("Not ......................");
    n = NOT(t);
    fmt.Println("NOT TRUE: ", n);
    n = NOT(f);
    fmt.Println("NOT FALSE: ", n);
    fmt.Println("And ......................");
    a = AND(t, t);
    fmt.Println("TRUE AND TRUE: ", a);
    a = AND(t, f);
    fmt.Println("TRUE AND FALSE: ", a);
    a = AND(f, t);
    fmt.Println("FALSE AND TRUE: ", a);
    a = AND(f, f);
    fmt.Println("FALSE AND FALSE: ", a);
    fmt.Println("Or .......................");
    o = OR(t, t);
```

```
    fmt.Println("TRUE OR TRUE: ", o);
    o = OR(t, f);
    fmt.Println("TRUE OR FALSE: ", o);
    o = OR(f, t);
    fmt.Println("FALSE OR TRUE: ", o);
    o = OR(f, f);
    fmt.Println("FALSE OR FALSE: ", o);
    fmt.Println("Xor ......................");
    x = XOR(t, t);
    fmt.Println("TRUE XOR TRUE: ", x);
    x = XOR(t, f);
    fmt.Println("TRUE XOR FALSE: ", x);
    x = XOR(f, t);
    fmt.Println("FALSE XOR TRUE: ", x);
    x = XOR(f, f);
    fmt.Println("FALSE XOR FALSE: ", x);
    fmt.Println("Imp ......................");
    i = IMP(t, t);
    fmt.Println("TRUE IMP TRUE: ", i);
    i = IMP(t, f);
    fmt.Println("TRUE IMP FALSE: ", i);
    i = IMP(f, t);
    fmt.Println("FALSE IMP TRUE: ", i);
    i = IMP(f, f);
    fmt.Println("FALSE IMP FALSE: ", i);
}
```

Output:

```
Constants .................
TRUE:  1
FALSE:  0
Not .......................
NOT TRUE:  0
NOT FALSE:  1
And .......................
TRUE AND TRUE:  1
TRUE AND FALSE:  0
FALSE AND TRUE:  0
FALSE AND FALSE:  0
Or ........................
TRUE OR TRUE:  1
TRUE OR FALSE:  1
FALSE OR TRUE:  1
FALSE OR FALSE:  0
Xor .......................
TRUE XOR TRUE:  0
TRUE XOR FALSE:  1
FALSE XOR TRUE:  1
FALSE XOR FALSE:  0
Imp .......................
TRUE IMP TRUE:  1
TRUE IMP FALSE:  0
FALSE IMP TRUE:  1
FALSE IMP FALSE:  1
```

EXAMPLES 3 – C LANGUAGE

Logical Functions

```
/* Lambda ..............................................
TRUE = λx.λy.x
FALSE = λx.λy.y
NOT = λx.x FALSE TRUE
AND = λx.λy. x y FALSE
OR = λx.λy. x TRUE y
XOR = λx.λy. x (y FALSE TRUE) y
IMP = λx.λy. x y TRUE
..................................................... */
```

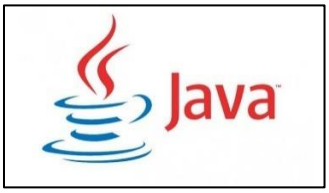
```
/* Constants ........................... */
#define True 1
#define False 0

typedef int (*func_ptr)(int, int);

/* Functions for Calls ................. */

int call_func(func_ptr f, int x, int y) {
    return f(x, y);
}

int call_default(func_ptr f) {
    return call_func(f, True, False);
}

/* Functions for Constants …............ */

int TRUE(int x, int y) {
    return x;
}

int FALSE(int x, int y) {
    return y;
}

/* Unary functions ..................... */

int NOT(func_ptr x) {
    return call_func(x, False, True);
}
```

```
/* Binary functions .................... */

int AND(func_ptr x, func_ptr y) {
    return call_func(x,
        call_default(y),
        call_default(FALSE));
}

int OR(func_ptr x, func_ptr y) {
    return call_func(x,
        call_default(TRUE),
        call_default(y));
}

int XOR(func_ptr x, func_ptr y) {
    return call_func(x,
        call_func(y,
            call_default(FALSE),
            call_default(TRUE)),
        call_default(y));
}

int IMP(func_ptr x, func_ptr y) {
    return call_func(x,
        call_default(y),
        call_default(TRUE));
}

/* Test code ........................... */

int boolean() {
    func_ptr t, f;
```

```
    int T, F, n, a, o, x, i;

    printf("Constants .................\n");
    t = TRUE;
    T = call_default(t);
    printf("TRUE: %d\n", T);
    f = FALSE;
    F = call_default(f);
    printf("FALSE: %d\n", F);
    printf("Not .......................\n");
    n = NOT(t);
    printf("NOT TRUE: %d\n", n);
    n = NOT(f);
    printf("NOT FALSE: %d\n", n);
    printf("And .......................\n");
    a = AND(t, t);
    printf("TRUE AND TRUE: %d\n", a);
    a = AND(t, f);
    printf("TRUE AND FALSE: %d\n", a);
    a = AND(f, t);
    printf("FALSE AND TRUE: %d\n", a);
    a = AND(f, f);
    printf("FALSE AND FALSE: %d\n", a);
    printf("Or ........................\n");
    o = OR(t, t);
    printf("TRUE OR TRUE: %d\n", o);
    o = OR(t, f);
    printf("TRUE OR FALSE: %d\n", o);
    o = OR(f, t);
    printf("FALSE OR TRUE: %d\n", o);
    o = OR(f, f);
    printf("FALSE OR FALSE: %d\n", o);
    printf("Xor .......................\n");
    x = XOR(t, t);
    printf("TRUE XOR TRUE: %d\n", x);
    x = XOR(t, f);
    printf("TRUE XOR FALSE: %d\n", x);
    x = XOR(f, t);
    printf("FALSE XOR TRUE: %d\n", x);
    x = XOR(f, f);
    printf("FALSE XOR FALSE: %d\n", x);
    printf("Imp .......................\n");
    i = IMP(t, t);
    printf("TRUE IMP TRUE: %d\n", i);
    i = IMP(t, f);
    printf("TRUE IMP FALSE: %d\n", i);
    i = IMP(f, t);
    printf("FALSE IMP TRUE: %d\n", i);
    i = IMP(f, f);
    printf("FALSE IMP FALSE: %d\n", i);
    return 0;
}
```

3.2. Additional Examples

Language 1 – JVM Intermediate Language (GPL)



Compiled code to be executed by the Java Virtual Machine is represented using a hardware and operating system-independent binary format, typically (but not necessarily) stored in a file, known as the class file format.

The class file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

Types and JVM

Most of the instructions in the Java virtual machine instruction set encode type information about the operations they perform.

- For instance, the iload instruction loads the contents of a local variable, which must be an int, onto the operand stack. The fload instruction does the same with a float value. The two instructions may have identical implementations but have distinct opcodes.

- For most typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter:
  - i for an int operation, l for long, s for short, b for byte,
  - c for char, f for float, d for double, a for reference.
- **Note:** the focus is the instruction for integers and references (due to Strings).

Given the JVM's one-byte opcode size, encoding types into opcodes places pressure on the design of its instruction set.

- If each typed instruction supported all the Java virtual machine's runtime data types, there would be more instructions than could be represented in a byte.
- Instead, the instruction set of the JVM provides a reduced level of type support for certain operations.
- In other words, the instruction set is intentionally not orthogonal.
- Separate instructions can be used to convert between unsupported and supported data types as necessary.

Type support in the Java Virtual Machine instruction set

opcode	byte	short	int	long	float	double	char	reference
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		

f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Basic JVM Keywords

- Integer and reference keywords: { iconst, iload, istore, iinc, iaload, iastore, iadd, isub, imul, idiv, irem, ineg, ishl, ishr, iushr, iand, ior, ixor, if_icmpOP, ireturn, aconst, aload, astore, aaload, aastore, if_acmpOP, areturn, dup, new, sipush, return, getstatic, ldc, invokevirtual, invokespecial }.
- ***Note**: Due to complexity of JVM-IS, the focus here is only with integer operations and references (that can be applied to Strings) or additional types/classes operations.*

- References:
  - Classical:
    - https://docs.oracle.com/javase/specs/jvms/se6/html/Overview.doc.html
    - https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html
  - Updated:
    - https://docs.oracle.com/javase/specs/jvms/se18/html/index.html

EXAMPLES – JAVA

Hello World

ORIGINAL CODE:

```java
public class Hello {
	public static void main(String[] args) {
		System.out.println("Hello");
	}
}
```

BASIC DISASSEMBLER (javap -c):

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #16                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #22                 // String Hello
       5: invokevirtual #23                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
       8: return
}
```

Sphere Volume

ORIGINAL CODE:

```
public class VolumeInt {
	public static void main(String[] args) {
		int PI = 314, r = 5, vol;
		vol = 4 * PI * r * r * r / 300;
		System.out.println(vol);
	}
}
```

BASIC DISASSEMBLER (javap -c):

```
Compiled from "VolumeInt.java"
public class VolumeInt {
  public VolumeInt();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: sipush        314
       3: istore_1
       4: iconst_5
       5: istore_2
       6: iconst_4
       7: iload_1
       8: imul
       9: iload_2
      10: imul
      11: iload_2
      12: imul
      13: iload_2
      14: imul
      15: sipush        300
      18: idiv
      19: istore_3
      20: getstatic     #16                 // Field java/lang/System.out:Ljava/io/PrintStream;
      23: iload_3
      24: invokevirtual #22                 // Method java/io/PrintStream.println:(I)V
      27: return
}
```

Basic Data types

ORIGINAL CODE:

```
public class SofiaData {
	public static void main(String[] args) {
		int area = 492;
		String countryName = "Bulgary";
```

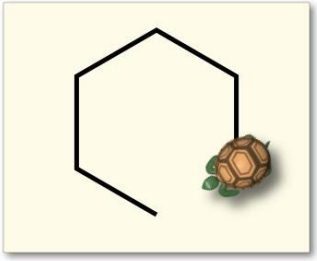
```
            System.out.println(countryName + "," + area);
        }
}
```

BASIC DISASSEMBLER (javap -c):

```
Compiled from "VolumeInt.java"
public class VolumeInt {
  public VolumeInt();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: sipush        314
       3: istore_1
       4: iconst_5
       5: istore_2
       6: iconst_4
       7: iload_1
       8: imul
       9: iload_2
      10: imul
      11: iload_2
      12: imul
      13: iload_2
      14: imul
      15: sipush        300
      18: idiv
      19: istore_3
      20: getstatic     #16                 // Field java/lang/System.out:Ljava/io/PrintStream;
      23: iload_3
      24: invokevirtual #22                 // Method java/io/PrintStream.println:(I)V
      27: return
}
```

Language 2 – LOGO (DSL)

LOGO BASICS



Logo is a programming language developed in the 1960's by Seymour Papert. Papert was the developer of an original and highly influential theory on learning called "*constructionism*" which could be summarized with the expression: "*learning by doing*".

We can see an interesting implementation in the following site: https://xlogo.tuxfamily.org/. A complete manual can be found in: http://downloads.tuxfamily.org/xlogo/downloads-en/manual-en.pdf.

Primitives

- forward number
  - Example: fd 50
  - **Explanation:** Moves the turtle forward number of steps in the direction it is currently facing.
- back number
  - Example: bk 100
  - **Explanation:** Moves the turtle backwards number of steps in the direction it is currently facing.
- right number
  - Example: bk 100
  - **Explanation:** Turns the turtle towards the right in relation to the direction it is currently facing.
- left number
  - Example: lt 45
  - **Explanation:** Turns the turtle towards the left in relation to the direction it is currently facing.
- clearscreen
  - Example: cs
  - **Explanation**: Erases the drawing area.
- showturtle
  - Example: st
  - **Explanation:** The turtle is visible on screen.
- hideturtle
  - Example: ht
  - **Explanation:** The turtle is invisible (drawing is faster).
- penup
  - Example: pu
  - **Explanation**: The turtle won't draw a line when it moves.
- pendown
  - Example: pd
  - **Explanation**: The turtle will draw a line when it moves.

Additional commands

- repeat integer list
  - Example: repeat 5[fd 50 rt 45]
  - **Explanation**: Repeat instructions contained in the list.

- to funcName [scope]
  - Example: to square
    repeat 4 [fd 100 rt 90]

end

- **Explanation**: Creates a procedure (function) to perform one action.

- setposition list
  - Example: setpos [100, -250]
  - **Explanation**: Moves the turtle to a specific position.

- wait number
  - Example: wait 60
  - **Explanation**: Pause the program during the number of 60th seconds.

- penerase
  - Example: penerase
  - **Explanation**: When the turtle moves, it erases all it encounters instead of drawing.

- penpaint
  - Example: penpaint
  - **Explanation**: Returns to classic mode. The turtle draws lines when it moves.

Keywords summarization (recommended for implementation):

- Basic keywords: { FD, RT, LT, BK, CS, PU, PD, HOME, SETXY, MAKE, PRINT, REPEAT, WHILE, IF, IFELSE, TO, END }

- References:
  - Online language: https://www.calormen.com/jslogo/
  - Some examples: https://xlogo.tuxfamily.org/en/index-en.html
    - Downloadable code: https://sourceforge.net/projects/xlogo4schools/

EXAMPLES – LOGO LANGUAGE

Hello World

```
cs
print "Hello"
```

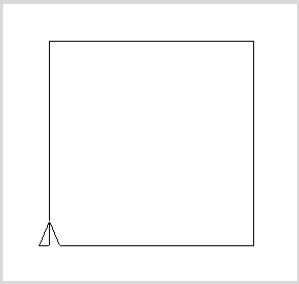
OUTPUT:

```
Hello
```

Basic Commands (Square)

```
cs
repeat 4[fd 200 rt 90]
```

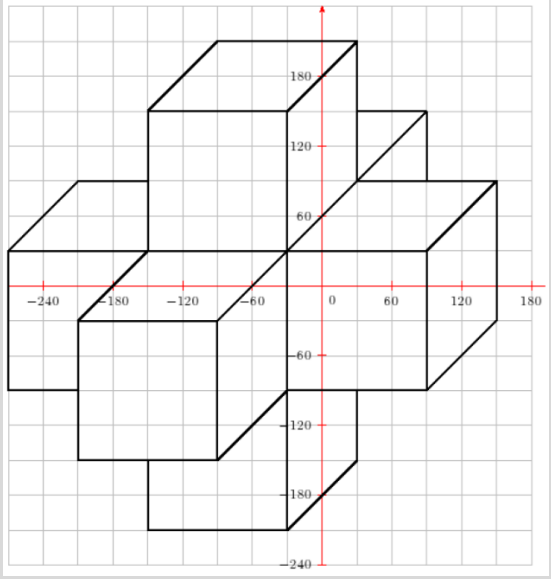
OUTPUT:



Additional Commands (Supercube)

```
cs
pu
setpos[ -30 150]
pd
setpos[-150 150]  setpos[-90 210]   setpos[30 210]
setpos[-30 150]   setpos[-30 -210]  setpos[30 -150]
setpos[30 -90]    setpos[-30 -90]   setpos[90 -90]
setpos[90 30]     setpos[-270 30]   setpos[-270 -90]
setpos[-210 -90]  setpos[-210 -30]  setpos[-90 -30]
setpos[-90 -150]  setpos[-210 -150] setpos[-210 -30]
setpos[-150 30]   setpos[-30 30]    setpos[-90 -30]
setpos[90 150]    setpos[30 150]    setpos[30 210]
setpos[30 90]     setpos[90 90]     setpos[90 150]
setpos[90 90]     setpos[150 90]    setpos[150 -30]
setpos[90 -90]    setpos[90 30]     setpos[150 90]
pu
setpos[-150 30]
pd
setpos[-150 150]  setpos[-150 90]   setpos[-210 90]
setpos[-270 30]
pu
setpos[-90 -150]
pd
setpos[-30 -90]
pu
setpos[-150 -150]
pd
setpos[-150 -210] setpos[-30 -210]
```

OUTPUT:



Language 3 – BRAZIL Assembly (DSL)

Basic Idea

This example illustrates another fictional language: "Brazil" (extension: ".BRZ").

***Notes**:*

- *This idea is to illustrate that you **can create** languages to be used by different purposes – high or low level.*
- *Even being considered as "**assembly**", this language is in fact a kind of **DSL** since the context is for low level machines.*

- Basic datatype:

* Integers:
  - Using hexadecimal signed values
  - Ex: LITERAL / CONSTANTS (2 bytes):
    $(0000)_{16}$ $(0)_{10}$ ... $(FFFF)_{16}$ $(65535)_{10}$.
  - *Remember: Hex: 0=0... , A=10, ... F=15*

- Variables:

* Only using 16 Registers:
  - `R0...RF` (Registers: R_n, n=0...F)
  - Note: They use the prefix '@' $(0040)_{16}$

- Comments (single line):

* Starts with # = $(0023)_{16}$

- Basic operations:

**** Input / Output:***

* **SC** (CodOp = 1): Scan: keyboard

    * **DP** (CodOp = 2): Display: screen
* ***Memory manipulation:***
    * **LD** (CodOp = 3): Load (register / constant)
    * **ST** (CodOp = 4): Store (register)
* ***Arithmetic:***
    * **AD** (CodOp = 5): Integer Addition
    * **SU** (CodOp = 6): Integer Subtration
    * **MU** (CodOp = 7): Integer Multiplication
    * **DI** (CodOp = 8): Integer Division

* ***Logical:***
    * **AN** (CodOp = 9): And operator
    * **OR** (CodOp = A): Or operator
    * **NT** (CodOp = B): Not operator
* ***Functions:***
    * **BF**: (CodOp = C): Beginning of Function
    * **EF**: (CodOp = D): End Function
        * Return value (0000 = NULL)
    * *Syntax:*
        * Names: using '_' for function names
        * Scope: Using ":"and finishing with EF.
    * *Template:*

```
BF _FUNCNAME_ PARAM_LIST:
  #COMMANDS
EF REG
```

* ***Extra commands (controls):***
    * **ST**: (CodOp = 0): Start Operation
    * **LF**: (CodOp = E): Load function
        - Syntax:

```
LF RET _NAME_ PARAMS
```

    * **HT**: (CodOp = F): Halt (end of execution).

> ***NOTE:***
> *In this hypothetical language, functions have indexes to label their names (until 15). The main program (not named) uses the index 0.*

EXAMPLES – BRAZIL

- Sample 1:

```
LD R0
ST 0001
# Intermed. rep.:
3 @0
4 0001
# Binary:
01101000
1000001
```

Functions samples (note the indexes)

```
# BRZ Code (Simple addition):
BF _ADD_ R0 R1:
LD R0
AD R1
ST R2
EF R2
# Intermed. rep.:
# Func Index
1: _ADD_
# Representation:
C 1 @0 @1
3 @0
5 @1
4 @2
D @2
```

```
# BRZ Code (Multiplication by 2):
BF _MULTBYTWO_ R1:
LD R1
MU 0002
ST R1
EF R1
# Func Index
2: _MULTBYTWO_
# Representation:
C 2 @1
7 0002
5 @1
4 @1
D @1

# Hello World - Func 3
# BRZ Code:
BF _HELLO_:
DP 48 # Letter H
DP 45 # Letter E
DP 4C # Letter L
DP 4C # Letter L
DP 4F # Letter O
EF 00
# Invoking function:
ST
LF 00 _HELLO_
```

```
HT

# Sphere volume - Func 4
BF _VOLUME_ R0:
LD R0
MU R0
MU R0
MU 0004
MU 7AAB # 31400
DI 7530    # 30000
ST R0
EF R0
# Invoking function:
ST
SC R2
LF R3 _VOLUME_ R2
DP R3
HT
```

Language 4 – Python (GPL)

EXAMPLES – PYTHON

Hello World

```
# Ex 1: This program prints Hello, world!
def main():
    print('Hello, world!')
main()
```

Output:

```
Hello, world!
```

Sphere Volume

```
# Ex 2: This program calculates the volume of a sphere
def volume():
    pi = 3.1415926535897931
    radian = float(input('Radius of sphere: '))
    volume = (4/3) * (pi * radian ** 3)
    print('Volume: ', volume)
volume()
use std::io;
fn main() {
    let pi: f64 = 3.1415926535897931;
    let mut input = String::new();
    println!("Radius of sphere: ");
    io::stdin().read_line(&mut input).expect("Failed to read input");
    let radian: f64 = input.trim().parse().expect("Invalid input");
    let volume: f64 = (4.0/3.0) * (pi * radian.powf(3.0));
    println!("Volume: {}", volume);
}
```

Output:

```
Radius of sphere: 2
```

```
Volume:  33.510321638291124
```

Some Data types

```
# Ex 3: SOFIA city #
def main():
      countryName = 'Bulgary'
      print(type(countryName))
      foundationYear = -7000
      print(type(foundationYear))
      area = 492.0
      print(type(area))
      print(countryName, ' was founded in: ', foundationYear, \
      ' and has ', area, ' km2 area')
main()
```

Output:

```
<class 'str'>
<class 'int'>
<class 'float'>
Bulgary  was founded in:  -7000  and has  492.0  km2 area
```

Logical Functions

```
# Example: Lambda calculus .....................

def true(x, y):
    #TRUE = λx.λy.x
    return x

def false(x, y):
    #FALSE = λx.λy.y
    return y

def Not(x):
    #NOT = λx.x FALSE TRUE
    return x(false, true)

def And(x, y):
    #AND = λx.λy. x y FALSE
    return x(y, false)

def Or(x, y):
    #OR = λx.λy. x TRUE y
    return x(true, y)

def Xor(x, y):
    #XOR = λx.λy. x (y FALSE TRUE) y
    return x(y(false, true), y)

def Imp(x, y):
    #IMPLIES = λx.λy. x y TRUE
    return x(y, true)

# Tests
```

```
print('Testing..................')
logic1=true
print('Logic1:', logic1.__name__)
logic2=false
print('Logic2:', logic2.__name__)
logic3=Not(false)
print('Logic3:', logic3.__name__)
logic4=Not(true)
print('Logic4:', logic4.__name__)
logic5=Or(true,false)
print('Logic5:', logic5.__name__)
logic6=And(true,false)
print('Logic6:', logic6.__name__)
logic7=Xor(true,false)
print('Logic7:', logic7.__name__)
logic8=Imp(true,false)
print('Logic8:', logic8.__name__)

# (End of demo)  ........................
```

Output:

```
> python demoLambda.py
Testing..................
Logic1: true
Logic2: false
Logic3: true
Logic4: false
Logic5: true
Logic6: false
Logic7: true
Logic8: false
```

Algonquin College
Fall, 2023

Last update[10]: 8th Sep 2023.

[10] Update versions:

- Sep 1st 2023: Document creation
- Sep 8th 2023: Inclusion of **Go Language** examples (Section Lang 3 in "Part III – Language Specification").