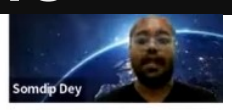
Introduction to languages and compilers

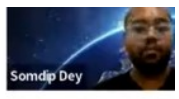
Programming Languages



- **High Level Languages**: Easy for programmers to write and reason with.
- **Low Level Languages**: Closer to what can be interpreted directly by hardware.
- **Machine Code**: Lowest level, and hardware specific.
  We need some way to translate from high-level (abstract) to low level (concrete) languages, while:
  - keeping the intended meaning.
  - giving no "unexpected" outcomes.
  - producing clear error messages if things go wrong.

Source: https://www.youtube.com/playlist?list=PLwPMc5UhSxNKXsj_52dCmTJdd1qbF9f65

Implementing Programming Languages



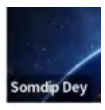
- **Strategy 1**: Interpreter - program that runs programs.
- **Strategy 2**: Compiler - program that translates program into machine code (interpreted by machine).
- **Modern trend is hybrid**:
  - Compilers that produce virtual machine code for bytecode interpreters.
  - “Just-In-Time” (JIT) compilers interpret parts of program, compile other parts during execution.

Brief History of Languages

- Initially, programs “hard-wired” or entered electro-mechanically
  - punched-card-handling machines.
- Next, stored-program machines
  - programs encoded as numbers (machine code) and stored as data.

Machine code	Assembly code	Description
001 1 000010	LOAD #2	Load the value 2 into the Accumulator
010 0 001101	STORE 13	Store the value of the Accumulator in memory location 13
001 1 000101	LOAD #5	Load the value 5 into the Accumulator
010 0 001110	STORE 14	Store the value of the Accumulator in memory location 14
001 0 001101	LOAD 13	Load the value of memory location 13 into the Accumulator
011 0 001110	ADD 14	Add the value of memory location 14 to the Accumulator
010 0 001111	STORE 15	Store the value of the Accumulator in memory location 15
111 0 000000	HALT	Stop execution

Brief History of Languages



- 1953: IBM developed the 701; all programming done in assembly.
- Problem: Software costs > hardware costs!
- John Backus: “Speedcoding” made a 701 appear to have floating point.
  - The first high-level programming language.
  - Easier to write programs (more “expressive”).
  - But the interpreter ran 10–20 times slower than native code.

Example of Machine Code

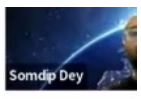
```
; The translation into LC-2 Machine Code
0011 0000 0000 0000          ; load at x3000
X3000 0010 001 0 0000 0110   ; LD R1, x006
x3001 0110 010 001 000000    ; LDR R2, R1, #0
x3002 0000 010 0 0000 010    ; BRz x005
;
; repeating statements go here ;
x3003 0001 001 001 1 00001   ; ADD R1,R1,#1
x3004 0000 111 0 0000 0001   ; BRnzp x001
x3005 1111 0000 0010 0101    ; HALT
```

After ; are comments

Hello, World! in Assembly

```
extern printf                ; the C printf function, to be called
    section .data            ; Data section, initialized variables
msg: db "Hello, world!", 0   ; C string terminates with 0
fmt: db "%s", 10, 0          ; The printf format, "\n",'0'
    section .text            ; Begin code section
    global main              ; the standard gcc entry point

main:                        ; the program label for the entry point
    push rbp                 ; set up stack frame, must be aligned
    mov rdi,fmt
    mov rsi,msg
    mov rax,0                ; can also be: xor rax,rax
    call printf              ; Call C printf function
    pop rbp                  ; restore stack
    mov rax,0                ; normal, no error, return value
    ret                      ; return
```

FORTRAN



- Also due to John Backus (1954–57).
- Revolutionary idea at the time: convert high-level to assembly.
- Called “automatic programming” at the time. Some thought it impossible.
- Wildly successful: language could cut development times from weeks to hours; produced machine code almost as good as hand-written.
- By 1958, 50% of programs written in Fortran.
- Start of extensive theoretical work (and Fortran is still with us!).

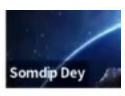
After FORTRAN

- Lisp, late 1950s: dynamic, symbolic data structures. Still in use today.
- Algol 60: Europe's answer to FORTRAN: modern syntax, block structure, explicit declaration.
- COBOL: late 1950's (and still with us). Business- oriented. Introduces records (structs).

The Language Explosion

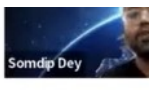
- APL (arrays), SNOBOL (strings), FORMAC (formulae), and many more.
- 1967-68: Simula 67, first "**object-oriented**" language.
- 1968: "Software Crisis" announced. Trend towards simpler, "cleaner" languages: Pascal, C

The 1970s



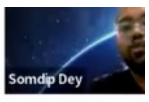
- Emphasis on "methodology": modular programming, CLU, Modula family.
- Mid 1970's: Prolog. Declarative logic programming.
- Mid 1970's: ML (Metalanguage) type inference, pattern-driven programming. (Led to Haskell, OCaml).
- Late 1970's: the US department of defence starts to develop Ada.

Into the Present



- Arguably, complexity increases with C++, C#, later versions of Java.
- Proliferation of little or specialised languages and scripting languages: HTML, PHP, Perl, Python, Ruby, Javascript, . . .
- Development of (online) repositories of development packages and libraries.
- Some reaction against (confusing) complexity of object orientation, and of memory management, as with *Go*, *Rust.*
- These have a focus on types, type safety, and more coherent/straightforward object models, and garbage collection.

A Sorting Algorithm



- It is enlightening to see how varied programming languages can be.
- Here we give an example of a sorting algorithm implementing in four different languages:
  1. Fortran
  2. Algol60
  3. APL
  4. Prolog

Example: FORTRAN


```
C FORTRAN (OLD-STYLE) SORTING ROUTINE
C
   SUBROUTINE SORT (A, N)
   DIMENSION A(N)
   IF (N - 1) 40, 40, 10
10    DO 30 I = 2, N
      L = I-1
      X = A(I)
      DO 20 J = 1, L
         K = I - J
         IF (X - A(K)) 60, 50, 50
C FOUND INSERTION POINT: X >= A(K)
50          A(K+1) = X
            GO TO 30
C ELSE, MOVE ELEMENT UP
60          A(K+1) = A(K)
20    CONTINUE
      A(1) = X
30 CONTINUE
40 RETURN
   END

C ---------------------------------
C MAIN PROGRAM
     DIMENSION Q(500)
100  FORMAT(I5/(6F10.5))
200  FORMAT(6F12.5)
     READ(5, 100) N, (Q(J), J = 1, N)
     CALL SORT(Q, N)
     WRITE(6, 200) (Q(J), J = 1, N)
     STOP
     END
```

Example: Algol 60

```
comment An Algol 60 sorting program;
procedure Sort (A, N)
  value N;
  integer N; real array A;
begin
  real X;
  integer i, j;
  for i := 2 until N do begin
    X := A[i];
    for j := i-1 step -1 until 1 do
      if X >= A[j] then begin
        A[j+1] := X; goto Found
      end else
        A[j+1] := A[j];
    A[1] := X;
  Found:
    end
  end
end Sort
```

Example: APL

```
⍝ An APL sorting program
▽ Z ← SORT A
      Z ← A [⍋A ]
▽
```

Example: Prolog

```
/* A naive Prolog sort */
/* permutation(A,B) iff list B is a
   permutation of list A. */
permutation(L, [H | T]) :-
   append(V,[H|U],L),
   append(V,U,W),
   permutation(W,T).
   permutation([], []).
/* ordered(A) iff A is in ascending order. */
ordered([]).
ordered([X]).
ordered([X,Y|R]) :- X <= Y, ordered([Y|R]).
/* sorted(A,B) iff B is a sort of A. */
sorted(A,B) :- permutation(A,B), ordered(B).
```

Problems to Address

- How to describe language clearly for programmers, precisely for implementers?
- How to implement description, and know you're right?
  (Automatic conversion of description to implementation)
- How to test?
- How to save implementation effort?
  - With multiple languages to multiple targets: can we re-use effort?
- How to make languages usable?
  - Handle errors reasonably
  - Detect questionable constructs
  - Compile quickly

Compiler Structure (Front-end)

↓ *design + editing*

Source Program

1. ↓ *lexical analysis*

Token Stream

2. ↓ *parsing*

Abstract Syntax Tree (AST)

3. ↓ *static semantic analysis*

Attributed/Annotated Syntax Tree

4. ↓ *optimisations*

Optimised Tree

5. ↓ *code generation*

Virtual Code

Compiler Structure (back-end)

Virtual Code

1. ↓ *code generation*

Real Machine Code

2. ↓ *optimisation*

Optimised Machine Code

Compiler Structure (Linker)

1. Optimised Machine Code + Other Object Code + Library Code

   ↓ *linking*

2. Executable File

General Structure (Summary)

- A sequence of translations that each:
  - Filter out errors.
  - Remove or put aside extraneous information.
  - Make data more conveniently accessible.
- The final target is to produce an equivalent, optimised program that can be executed by the intended machine.
- One Strategy: use tools that partially automate this procedure.

Summary of Stages

- Lexical Analysis
  - Recognise the words (one level above characters).
  - Generally not trivial.
- Divide program text into words or “tokens”.
  If x == y then z = 1; else z = 2;
- Parsing
  - Identify the “sentences” (statements, and program).
  - Produce the syntax structure (syntax tree).

Summary of Stages

- Static Semantic Analysis
  - Once parsed, we can then check the program is meaningful.
  - Full analysis of meaning is complex.
  - Compilers perform conservative analysis to check consistency.
    - Type checking.
    - Items defined before they are used.

Summary of Stages

- Optimisation
  - Make the program more efficient (smaller, faster).
  - Remove redundancy.
- Code generation
  - Translate into machine code (or an intermediate language).
  - Akin to translating between different human languages, which have different words and grammar.

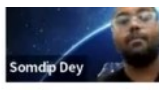
Intermediate Languages

- Typically there are two code generation steps:
  - an abstract, general purpose intermediate language;
  - the final target architecture.
- They progress from the most abstract (the original program) to the least abstract (machine instructions).
- A fully fledged compiler will typically perform optimisations both before and after code generation.
- Lower levels express more specific information, such as registers and memory layout, but obscure higher-level constructs.
- The different representations of the program make it easier to express, identify, and implement different kinds of optimisations.

Issues

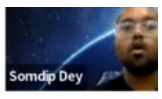
- There are many pitfalls in compiler design, such as how to handle program errors.
- The design of a language can have a radical impact on how easy it is to build a compiler, and optimise the generated code.
- There are criteria for good language design, such as ease of use and expressiveness.

Compiling Today



- Compilers have not changed much in general structure since early FORTRAN compilers.
- The biggest change is perhaps that increased processing power has altered the workflow for writing programs.
  - Previously this would have been “batch” orientated.
  - Now it tends to be interactive.
  - Such changes have an impact on the handling of errors, for example.
- The increase in computer power means that it is feasible to put more effort into compiler optimisation of programs.
  - Most mainstream compilers now put more effort into optimisation than any of the other phases.
  - Can be very important when (cross) compiling for an embedded system.

Why so many languages?

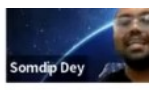


- Languages adopted, and adapted for a specific need.
- Task driven language adoption and adaption does not always lead to good language design.
- Motivates some to design “better” languages.
- Languages with large numbers of users will continue to be used (and rarely changed or improved to address technical flaws).
- New languages are easy to introduce for niche areas (and all too often poorly designed).

Language Design

- Criteria for good design are not easy to specify.
  - Should make it easy to write programs that are easy to read and which are reliable.
  - Should make it easy to find, and avoid errors, for example by using types and type-checking.
  - Should allow the programmer to focus on details at an appropriate level, through abstraction, and reuse, for example by using *functions, abstract data types, modules.*
- High-level languages also allow programmers to ignore irrelevant hardware specific details.
- Some approaches, such as object orientation, address several of these issues.

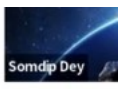
Summary



- The introduction of new languages, and new target platforms, means that work on **compilers is still required**.
- There are application areas where it is appropriate to develop **task specific languages** and language extensions and optimisations, with associated tools.
- Understanding compilers can help with the understanding of program behaviour.
- It is also an area in which **many fundamental issues** in computer science can be seen in action.

Compilation Stages

Introduction



- This module aims to learn the techniques of modern compilers.
- This lecture is a short tour of the compilation process. More details will be followed in this module.
- This lecture also provides a general overview of ANTLR's capabilities for lab sessions.

Compiler Structure (Front-end)

↓ *design + editing*

Source Program

1. ↓ *lexical analysis*

Token Stream

2. ↓ *parsing*

Abstract Syntax Tree (AST)

3. ↓ *static semantic analysis*

Attributed/Annotated Syntax Tree

4. ↓ *optimisations*

Optimised Tree

5. ↓ *code generation*

Virtual Code

Compiler Structure (back-end)

Virtual Code

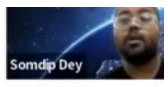
1. ↓ *code generation*

Real Machine Code

2. ↓ *optimisation*

Optimised Machine Code

Compiler Structure (Linker)



1. Optimised Machine Code + Other Object Code + Library Code

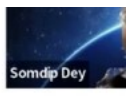
↓ *linking*

2. Executable File

Compiler Stages overview

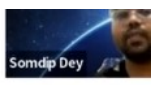
- Lexical analysis (Scanning)
- Syntax Analysis (Parsing)
- Static Semantic Analysis
- Intermediate Code Generation
- Code Generation
- Machine Independent Optimisation

Terminology



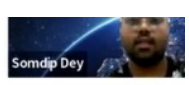
- Compiler:
  - a program that translates an *executable* program in one language into an *executable* program in another language.
  - we expect the program produced by the compiler to be better, in some way, than the original.
- Interpreter:
  - a program that reads an *executable* program and produces the results of running that program.
  - usually, this involves executing the source program in some fashion.

Grammar

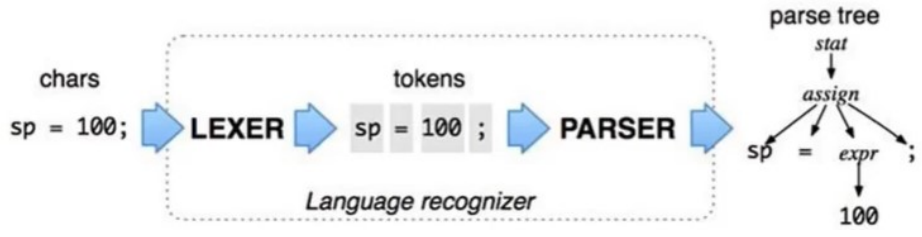


- If an application computes or "executes" sentences, we call that application an ***interpreter****, such as* calculators.
- If we're converting sentences from one language to another, we call that application a ***translator****,* such as C compilers.
- The interpreter or translator has to recognise all of the valid sentences, phrases, and subphrases of a particular language.
- Recognising a phrase means we can identify the various components.
- Programs that recognise languages are called *parsers* or *syntax analysers*.
- *Syntax* refers to the rules governing language membership.
- A grammar is just a set of rules, each one expressing the structure of a phrase.

Lexical Analysis and Parsing

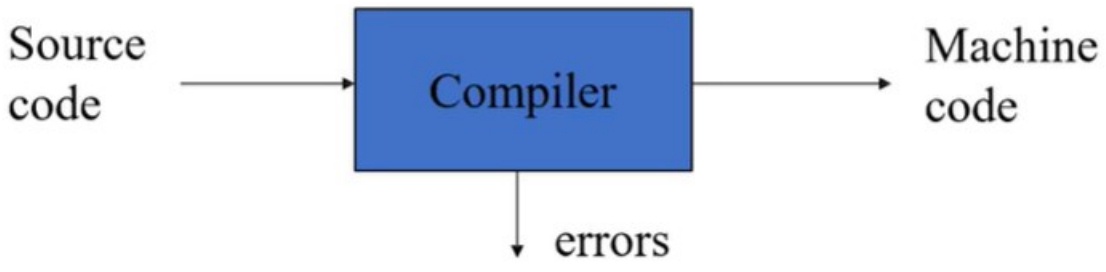


- Parsing is much easier if we break it down into two tasks or stages.
- The process of grouping characters into words or symbols (*tokens*) is called ***lexical analysis*** or simply *tokenising,* such as INT (integers), ID (identifiers), FLOAT (floating-point numbers).
- We call a program that tokenises the input a *lexer*.
- The second stage is the actual ***parser*** and feeds off of these tokens to recognise the sentence structure.



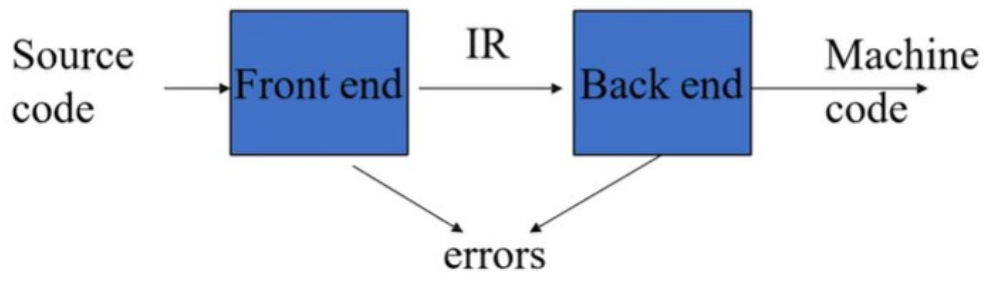
Abstract View

- Recognises legal (and illegal) programs.
- Generate correct code.
- Manage storage of all variables and code.
- Agreement on format for object (or assembly) code.



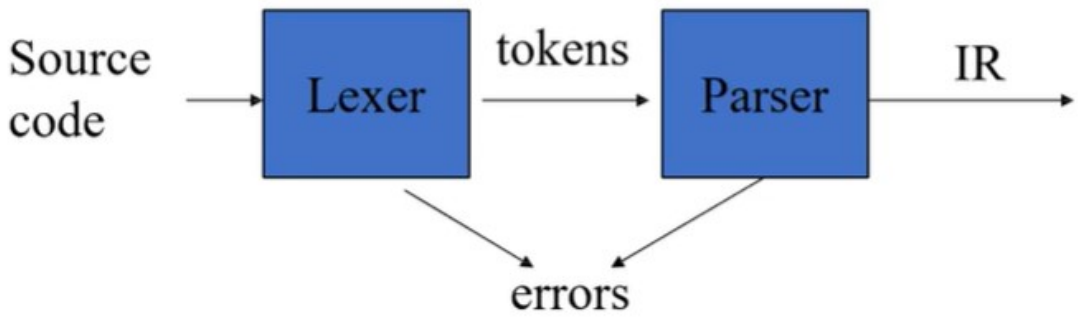
Front-end and Back-end Division

- Front-end maps legal code into Intermediate Representation (IR).
- Back-end maps IR onto target machine.



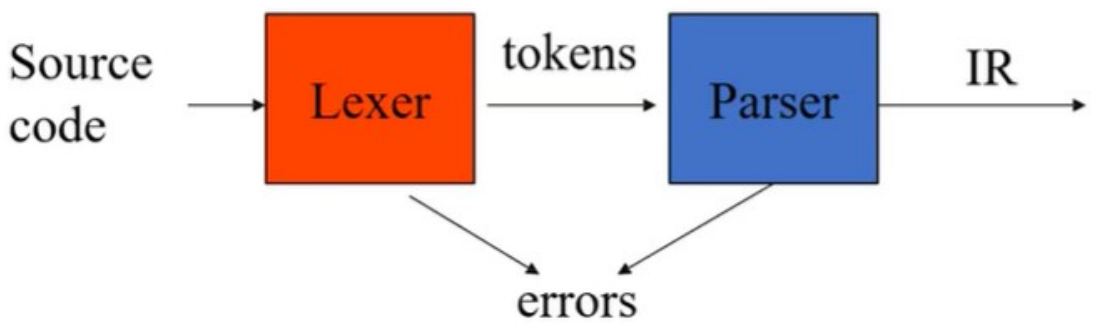
Front end

- Recognise legal code.
- Report errors.
- Produce IR.
- Preliminary storage maps.



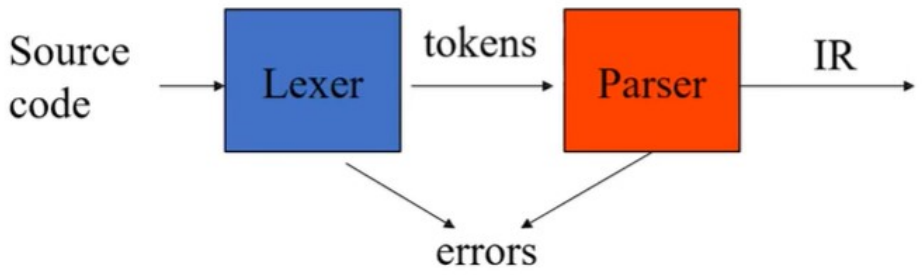
Front end

- Lexer:
  - Maps characters into tokens – the basic unit of syntax
    - x = x + y becomes <id, x> = <id, x> + <id, y>
  - Typical tokens: number, id, +, -, *, /, do, end
  - Eliminate white space (tabs, blanks, comments)



Front end

- Parser:
  - Recognise context-free syntax.
  - Construct IR.
  - Produce meaningful error messages.
  - Attempt error correction.
- There are parser generators which automates much of the work.



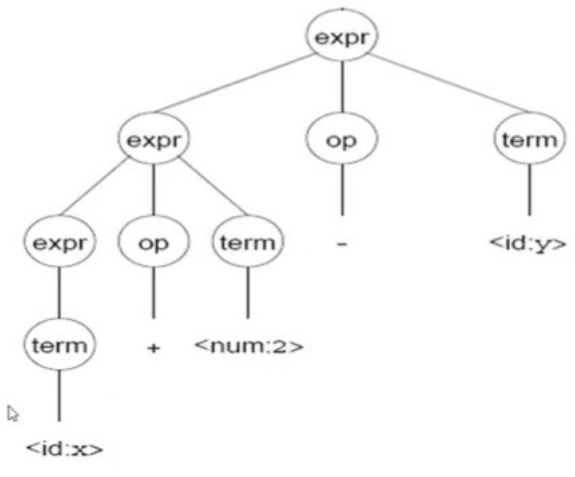
Front end

- Context free grammars are used to represent programming language syntax:

```
while sp = 100:
    sp = sp -1
```

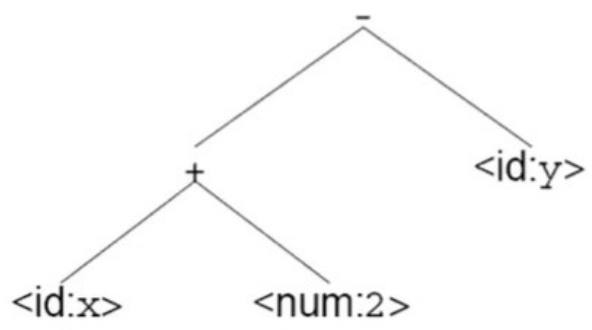
Front end

- A parser tries to map a program to the syntactic elements defined in the grammar.
- A parse can be represented by a tree called a parse or syntax tree.



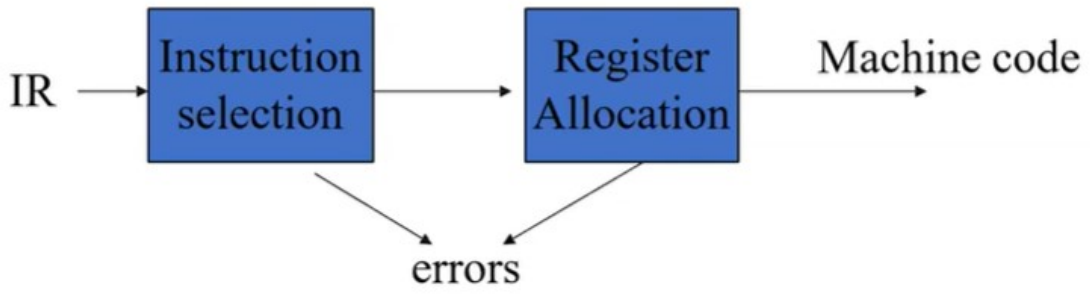
Front end

- A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST).
- AST is often used as IR between front end and back end.



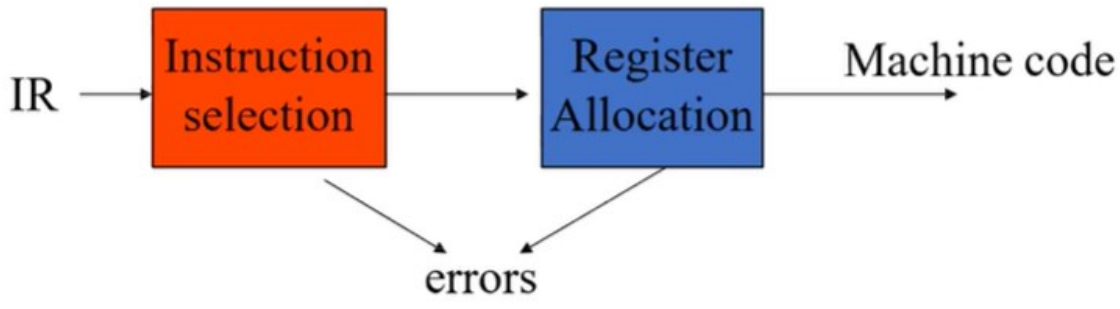
Back end

- Translate IR into target machine code.
- Choose instructions for each IR operation.
- Decide what to keep in registers at each point.
- Ensure conformance with system interfaces.



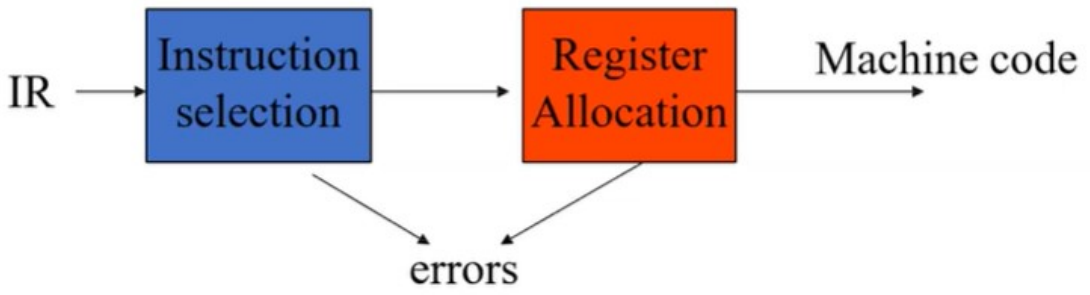
Back end

- Produce compact and fast code.
- Use available addressing modes.



Back end

- Have a value in a register when used.
- Limited resources.
- Optimal allocation is difficult.

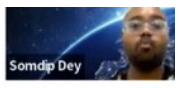


Introduction to ANTLR 4

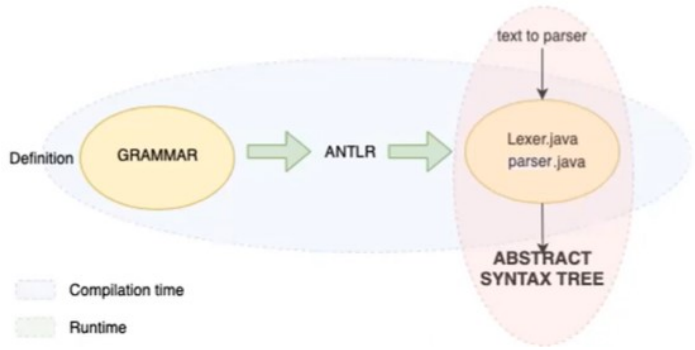
Antlr4

- Antlr stands for ANother Tool for Language Recognition by Terence Parr.
- ANTLR v4 is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files.
- It's widely used in academia and industry to build all sorts of languages, tools, and frameworks.
- From a formal language description called a *grammar*, ANTLR generates a parser for that language that can automatically build parse trees, which are data structures representing how a grammar matches the input.
- ANTLR also automatically generates tree "walkers" that you can use to visit the nodes of those trees to execute application-specific code.

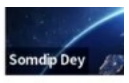
Development Process with Antlr



- Antlr can convert grammars into a parser and a lexer that can recognise sentences in the language described by the grammar.
- In **compilation time**, you define a grammar first. Antlr processes the grammar file and produce a lexer and a parser in Java.
- In **run time**, the text or sentence (input) in the language described by the grammar is fed into the lexer and parser. The output is an abstract syntax tree.

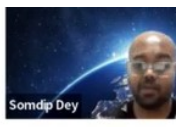


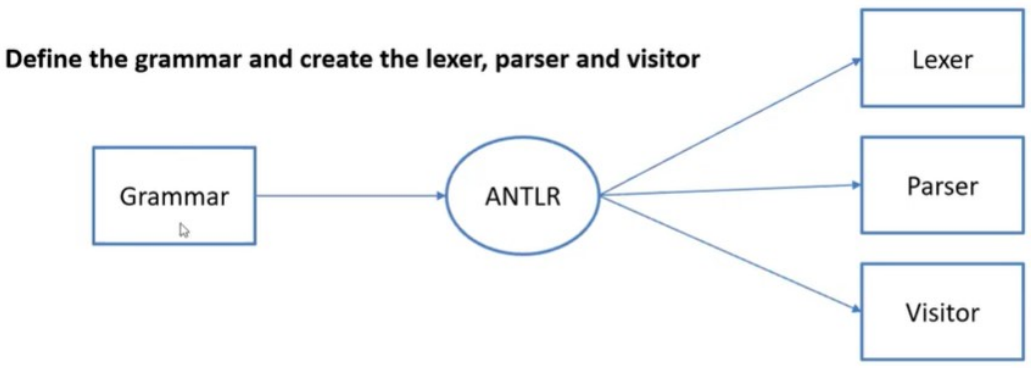
Antlr Outputs

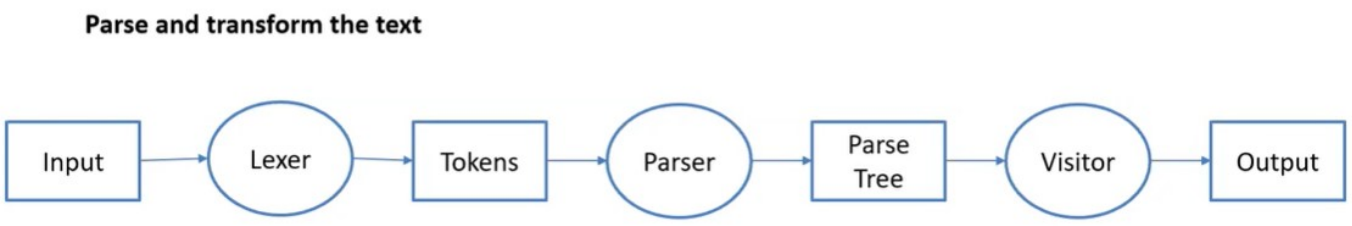


- Antlr4 includes a lexical analyser and a syntax analyser that will generate a parse tree (syntax tree).
- Antlr4 also generates runtime libraries: listeners and visitors for the parse tree that can:
  - evaluate the parse tree in the case of an interpreter;
  - generate intermediate and/or target code in the case of compliers.
- The generated code can be in **Java** (default), Javascript, Python, C++ and C#.

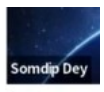
Antlr Workflow from Lab 1 (on Moodle)







Antlr4 Grammar File



- Let’s explore the structure of an Antlr4 grammar file: CSV - Comma Separated Value

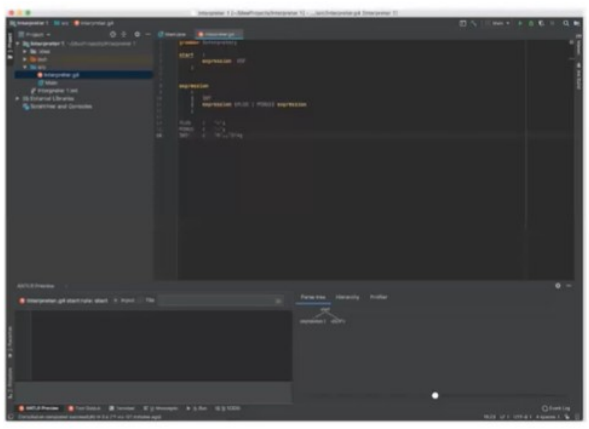
```
grammar csv;

list
: BEGL elems? ENDL
;

elems
: elem ( SEP elem )*
;

elem
: NUM
;

BEGL : '[';
ENDL : ']';
SEP  : ',';
NUM  : [0-9]+;
WS   : [ \t\r\n]+ -> skip;
```

Antlr4 Grammar File

- All Antlr4 commands end with a semicolon.
- Terminals (tokens) start with a capital letter.
- Non-terminals (rules) begin with a lower case letter.
- * represents zero or more repetitions.
- + represents one or more repetitions.
- A|B represents a choice (alternation) between A and B.
- [ABC] represents a choice of one from A, B or C.
- A? represents A|ε.
- Round brackets are used for grouping.
- Rules and tokens are matched from top to bottom, i.e. first rule or terminal that matches is used.
- Each grammar is given a name using the grammar command (line 1).
- File extension is always g4 (Hello.g4).

Testing a Grammar File in IntelliJ

- Click on "**start**" and then selecting **Test Rule Start**
- In the lower left box, type in a phrase like "[1,2,3,9,8]" and you can get the **parse tree**.
- You can also get a textual representation of the parse tree in LISP-style text form (*root children*).

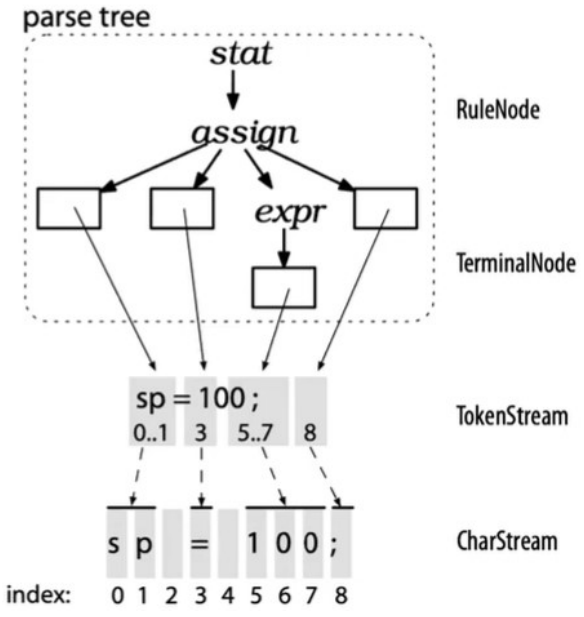


Generating Lexer and Parser

- Click on "**start**" and then selecting “**Generate ANTLR Recognizer”**
- The grammar csv.g4 will generate
  - a lexer called csvLexer.java
  - a parser called csvParser.java
  - a set of tokens csv.tokens
  - listener classes csvListener.java and csvBaseListener.java (by default)
  - visitor classes csvVisitor.java and csvBaseVisitor.java (if the -visitor option is selected).

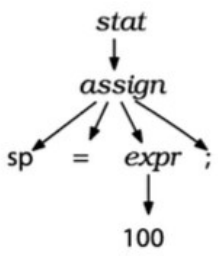
ANTLR Classes

- Lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree.
- The corresponding ANTLR classes are CharStream, Lexer, Token, Parser, and ParseTree.
- The “pipe” connecting the lexer and parser is called a TokenStream.
- RuleNode has methods such as getChild() and getParent().

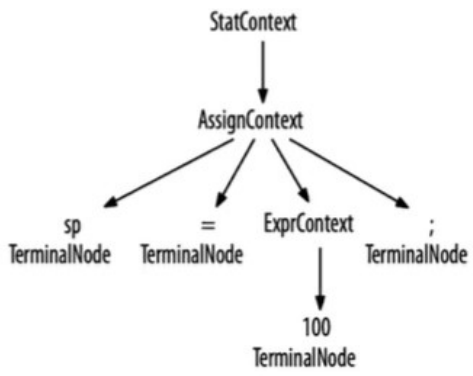


Antlr Context Objects

- ANTLR generates a RuleNode subclass for each rule.
  sp=100;
- AssignContext provides methods ID() and expr() to access the identifier node and expression.

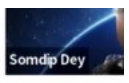


Parse tree



Parse tree node class names

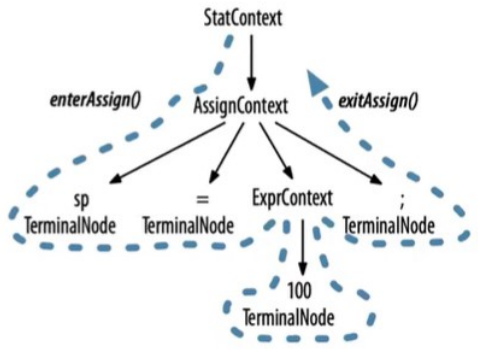
Parse Tree Listeners and Visitors



- Given a parse tree, we could write code by hand to perform a depth-first walk of the tree.
- We can take actions, such as computing results, updating data structures, or generating output, using tree-walking mechanisms.
- ANTLR provides support for two tree-walking mechanisms:
  - a parse-tree *listener* interface
  - a parse-tree *visitor* interface

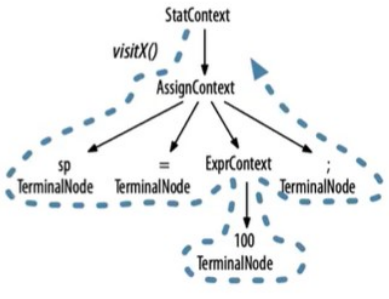
Antlr4 Listeners

- The listener mechanism provides two call-back methods for each rule X, an enterX method and an exitX method.
- These default methods do not do anything but can be overridden by the programmer.
- The parse tree walker performs a depth-first traversal of the parse tree.
- When the walker visits node X, the enterX method is invoked. The exitX method is invoked when the walker is leaving the X node.



Antlr4 Visitors

- When antlr4 is invoked with the -visitor option, a visit method will be generated for each node in the parse tree. For a node of type X, a visitX method is generated.
- Each visit method is parameterised by a **context object** and we use this context object to query the current node and to access its child nodes.
- If the grammar is called Y and the type of the node is X, then the context object has the type YParser.XContext.
- The programmer explicitly controls what further visit methods are invoked during a visit method.
- The default visit methods generated by Antlrv4 do not do anything and we need to extend them to add behaviour.



Writing Your Own Parser

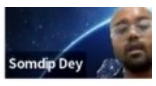
- You need a main function in order to implement your parser.

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CommonTokenStream;
import java.io.IOException;
import static org.antlr.v4.runtime.CharStreams.fromFileName;

public class Main {
    private final static String DIR = "input/";
    public static void main(String[] args) throws IOException {

        CharStream cs = fromFileName(DIR + "list01.txt");
         csvLexer lexer = new csvLexer(cs);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        csvParser parser = new csvParser(tokens);

         Integer sum = new EvalVisitor().visit(parser.list());
        System.out.println("sum = " + sum);*/
        String str = new PrettyVisitor().visit(parser.list());
        System.out.println(str);
    }
}
```

EvalVisitor.java

```
public class SumVisitor extends csvBaseVisitor<Integer> {

    @Override
    public Integer visitList(csvParser.ListContext ctx) {
        return ctx.elems() == null ? 0 : this.visitElems(ctx.elems());
    }

    @Override
    public Integer visitElems(csvParser.ElemsContext ctx) {
        int sum = 0;
        for (csvParser.ElemContext elemContext : ctx.elem()) {
            sum += this.visitElem(elemContext);
        }
        return sum;
    }

    @Override
    public Integer visitElem(csvParser.ElemContext ctx) {
        return Integer.valueOf(ctx.NUM().getText());
    }
}
```

Visitors vs Listener



- It depends on the problem and particular tasks.
- Visitors:
  - Visitors give you more control but put more responsibility on the programmer.
  - Visitors always return a value.
- Listeners:
  - Invoked by a walker that performs a depth-first traversal.
  - Particularly good at translations.
  - Entry and exit actions.

Summary

- Some key concepts: language, grammar, syntax tree (parse tree), token, lexer (tokeniser), parser.
- The computation process is divided into several stages: front-end and back-end.
- Front-end mainly consists of lexical analysis and syntax analysis.
- Antlr v4 is a powerful parser generator in Java by default.
- You can take actions, such as computing results, updating data structures, or generating output, using tree-walking mechanisms: Listener and Visitor.

Lexical analysis

Module on: How to build your own compiler

Example

- An example: the text of the program

```
if(i== j)
   x = 0; /* A comment */
else
   x= 1;
```

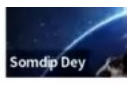
is just a sequence of characters:

```
\tif(i== j)\n\t\tx = 0; /* A comment */\n\telse\n\t\tx= 1;
```

- This needs to be partitioned into substrings corresponding to the tokens of the language.

Token

Tokens



- A token consists of a syntactic category plus “semantic” information conveyed by the underlying string.
- They convey the role of a given string.
- Then, parsing operates on syntactic categories rather than directly on words.
  - “Joe went to the store” and “Harry went to the beach” have same grammatical structure.
  - The parser relies on token identifies (an Identifier is treated differently to a Keyword, or Whitespace).

Tokens in Programs

- In general identifying words and their lexical category is called lexical analysis.
- For programming we tokenise by throwing away formatting and comments, identify where a token starts and ends, and assign appropriate token identifiers.
- These token identifiers correspond to syntactic categories plus semantic information.
- For programs, tokens will correspond to the programming language’s notion of a category, such as ***Identifier, Integer, Keyword, Whitespace***. Typically, these will have a compact internal representation.
- For programs, semantic information might correspond to the text of an identifier or numeral, or some representation of these values.

Example Program Tokens

- To illustrate, a programming language could use tokens defined as follows:
  - **Identifier** strings of letters or digits starting with a letter
  - **Integer** a non-empty string of digits
  - **Keywords** the specific strings used to express control structure, and declarations, such as `if, then, else, begin, end`.
  - **Whitespace** non-empty sequences of blanks, newlines and tabs.
  - **Comments**
  - Some **punctuation symbols** need to be tokenised. We might consider them to be like Keywords. They may be represented by tokens of the same form (the strings “(”, “)” being represented by the tokens “(” and “)”, or distinct tokens such as LBRACKET and RBRACKET).
- The output of lexical analysis will be a stream of tokens, some of which will include semantic information, such as the value of an Identifier.

Example Tokenisation

- Example

```
if(i== j)
   x = 0; /* A comment */
else
   x = 1;
```

might be tokenised as follows:

```
IF,LPAR,ID('i'),EQUALS,ID('j'),RPAR,ID(‘x'),ASSIGN,INTLI
T('0'),SEMI,ELSE,ID(‘x'),ASSIGN,INTLIT('1'),SEMI
```

Implementation

- An implementation of lexical analysis should
  - recognise substrings corresponding to tokens.
  - return the value of the lexeme (the substring) and its corresponding token.
  - Content that is not relevant to parsing (e.g. comments and whitespace) is discarded.

Ambiguity

- Is `if` the keyword "`if`" or the identifiers `i` and `f`?
- We might assume that we solve the problem by consider certain characters (whitespace and punctuation) as indicating token boundaries.

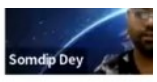
Ambiguity and Token Boundaries

- Unfortunately, in general we cannot rely on specific symbols in the input stream to identify token boundaries.
- FORTRAN: spaces are allowed anywhere for historic reasons.
- C++
  - Templates: `Foo<BAR>`
  - Stream syntax: `cin>>var`
  - Nested templates: `Foo<Bar<Baz>>`
- There is a potential conflict in assigning appropriate token for >> (and for <).

Ambiguity and Look-ahead

- In some cases, the only solution is to use "look-ahead" algorithms to ensure we don't misclassify a substring.
- For example, a lexical analyser for C or Java must read ahead after it sees the character ">". If the next character is "=" then ">" is part of the sequence ">=". Otherwise ">" itself forms the "greater than" operator.
- One-character read-ahead usually suffices. So a simple solution is to use a variable to hold the next input character.
- A lexical analyser can read ahead one character while it collects digits for numbers or characters for identifiers.
  - it reads past 1 to distinguish between 1 and 10.
- The lexical analyser reads ahead only when it must. An operator like "*" can be identified without reading ahead.

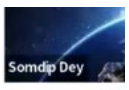
Recap



- **Identify** relevant lexemes (substrings) and their corresponding tokens (syntactic category) in the input (program).
- As we typically scan the input (program text) from left-to-right, we may need **lookahead** to resolve local ambiguity.
- We still need a way to describe the lexemes that correspond to a given token, and an **algorithm** for using those descriptions to match lexemes (with lookahead if needed).

Regular Expression and Grammar

Regular Expressions



- We need to be able to express/define the strings/lexemes that belong to each token.
- It turns out that individual words and lexemes of both programming languages and human languages can be characterised within a class of **regular languages**, equivalent in power to **regular expressions**.
- Regular expressions denote formal languages, which are sets of strings (of symbols from some alphabet).
- Regular expressions are easy to understand and allow efficient implementation.
- The theory of regular expressions, and the "languages" they characterise are useful.

Language Grammar

- In general, a language *L* can be defined as sequences of symbols drawn from an "alphabet" (Σ).
- "A language over Σ" is just an identified subset of "strings" drawn from elements of the alphabet Σ.
- In the case of lexical analysis for a compiler:
  - the alphabet Σ consists of the characters.
  - the sequences, or strings, are the lexemes of the programming language.

Grammars

- A given language may be characterised by a **grammar** G.
- We can say that the language generated or specified by G is $L(G)$.
- As we shall see later, grammars and the languages they specify, can be seen to fall into **different classes** corresponding to the computational power needed to parse strings of the language.
- **Tokens** fall into the class of so-called **regular languages**.
- These can be specified using Regular Expressions. (i.e. regular expressions provide a means of specifying the "grammar" needed to identify the lexemes of a program).

Regular Expressions as a Grammar Specifying a Lang

- Expression *R* denotes language $L(R)$ over alphabet Σ
  - $L(\epsilon) = L('') = \{!\,!\}$
  - If *c* is a character in Σ, $(c \in \Sigma)$, $L(c) = L('c') = \{!c!\}$
  - If $R_{"}$, $R_{\#}$ are regular expressions over Σ, then
    - $L(R_{"}R_{\#}) = \{x_{"}, x_{\#} | x_{"} \in L(R_{"}), x_{\#} \in L(R_{\#})\}$
    - $L(R_{"}|R_{\#}) = L(R_{"}) \cup L(R_{\#}) = \{x | x \in L(R_{!}) \; or \; x \in L(R_{"})$
    - $L(R^{*}) = L(\epsilon) \cup L(R) \cup L(RR) \cup \cdots = \cup_{\&'(} R^{\&}$
    - $L((R)) = L(R)$
- Precedence is "$*$" (highest), concatenation, union (lowest). Parentheses also provide grouping.

Regular Expression Abbreviations

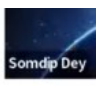
- There are some common abbreviations:
  - $L(R^{)}) = L(RR^*)$
  - $L(R?) = L(\epsilon|R)$
- And some implementation specific abbreviations:
  - Character lists, such as `[abcf-mxy]` in Java, Perl, or Python.
  - Character classes, such as `(dot)`, `\d`, `\s` in Java, Perl, Python.
  - Option A?, equivalent to A|ε.
- $a|b$ sometimes written $a + b$.
- [abcd] means (a|b|c|d)
- [a-d] means [abcd]
- a^+ means (aa*)

Examples

Let $\Sigma = \{a, b\}$

- The regular expression $a|b$ denotes the language $\{a, b\}$.
- $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. .Another regular expression for the same language is $aa|ab|ba|bb$.
- a^* denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \cdots\}$
- $(a|b)^*$ denotes the set of all strings consisting of zero or more instances of a or b that is all strings of a's or b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \cdots\}$.
  Another regular expression for the same language is $(a^*|b^*)^*$.
- $a|a^*b$ denotes the language: $\{a, b, ab, aab, aaab, \cdots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

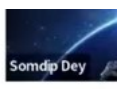
Examples



- C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers:

$$
\begin{aligned}
letter\_ &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id &\rightarrow letter\_ \; (\; letter\_ \mid digit \;)^*
\end{aligned}
$$

Examples



- Unsigned number (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`.
- The regular definition is a precise specification for this set of strings.
  - An *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string)
  - An *optionalExponent*, if not missing, is the letter E followed by an optional + or – sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match `1.`, but does match `1.0`.

$$
\begin{aligned}
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits &\rightarrow digit \; digit^* \\
optionalFraction &\rightarrow \texttt{.} \; digits \mid \epsilon \\
optionalExponent &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; digits \;) \mid \epsilon \\
number &\rightarrow digits \; optionalFraction \; optionalExponent
\end{aligned}
$$

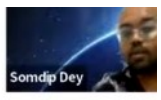
Recap

- We can now use regular expressions to define lexemes corresponding to individual tokens.
- We still need to find algorithms for identifying these lexemes in an input stream.
- Ideally it should be possible to generate such algorithms directly from a collection of regular expressions.
- There are tools that help with this.
- Such tools compile the specification into a program that implements the lexical analysis parser.

Sample Problems

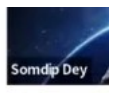
- Decimal numerals in C, Java.
- All numerals in C, Java.
- Floating-point numerals.
- Identifiers in C, Java.
- Comments in C++, Java.
- XHTML markups.
- Python bracketing.

Some Problem Solutions



- Decimal numerals in C, Java:
  `0|[1-9][0-9]*`
- All numerals in C, Java:
  `[1-9][0-9]+|0[xX][0-9a-fA-F]+|0[0-7]*`
- Floating-point numerals:
  `(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|[0-9]+[eE][-+]\d*`
  where `\d` means "`digit`", or `[0-9]`.
- Identifiers in C, Java:
  `[a-zA-Z][a-zA-Z0-9]*`
- Comments in C++, Java:
  `//.*|/\*([^*]|\*[^/])*\*/+`

Examples: Keywords



- **Keywords** `if, then and else`
  - **Regular expression** `'if'|'then'|'else'`

    where 'string' abbreviates for 's''t''r''i''n''g'.
- **Whitespace** non-empty sequence of blanks, tabs and newlines
  - **Regular expression** `(' '|'\n'|'\t')+`
- **Integers** a non-empty string of digits
  - **Regular expressions**

```
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
integers = digit digit*
```

- It can be clearer and more convenient to decompose lexeme definitions.

Examples: Identifiers

- **Identifiers:** strings of digits or letters starting with a letter
- **Regular expressions**

```
letter = 'A'|'B'|...'Z'|'a'|'b'|...'|'z'
identifier = letter(letter|digit)*
```

Examples: Email Address

- **Email addresses:** `name@subdomain.domain.tld`
- **Regular expressions**

```
Σ=letters+{@,.}   name=letter+
address=name'@'name(.name)+
```

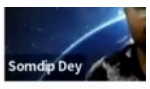
- Actually inadequate; misses many legitimate domain names.

Examples: Pascal Numbers

- **Pascal-like Numbers:** signed with an optional fractional part and optional exponent.
- **Regular expressions**

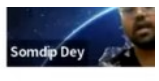
```
digit='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
digits=digit+
optsign=('+'|'-'|Ɛ)
optfract=('.' digits)|Ɛ
optexp=('E' optsign digits)|Ɛ
num=optsign digits optfract optexp
```

Combining token definitions



- All possible tokens can be defined by a single, structured regular expression.
- The names of the regular expression defined at the highest level then correspond to the token identifiers.

Summary



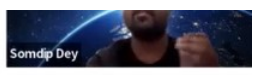
- Regular expressions are useful for defining languages.
- They are a specification, not an implementation.
- We need some way of checking whether a string *s* lies in the language defined by a regular expression `R`.
- That is we need to be able to compute `s ∈ L(R)`.

Finite Automata & Regular Expression

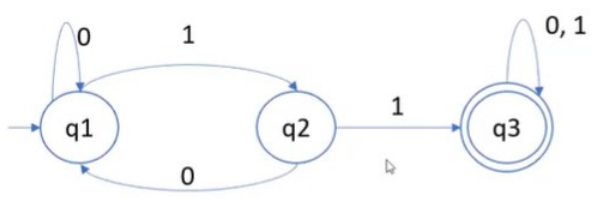
Introduction: Computability Theory

- Computability Theory (1930s – 150s)
- Goal: What is computable… or not
- For example, Developing specification for a sorting algorithm and having a program to verify if it meets the specification
- Models of Computation: Finite Automata
- Finite Automata is an abstract model of a computing machine with limited resources such as memory.

Example



M1



- States: q1, q2, q3
- Transitions: 1 →
- Start state: →○
- Accept state: ◎

- **Input**: Finite string
- **Output**: Accept or Reject
- **Computation process**:
  - Step 1: Begin at start state
  - Step 2: Read input symbol
  - Step 3: Follow corresponding transitions
  - Step 4: Accept if ends with accept state OR Reject if not
- **Examples**:
- 01101 -> Accept
- 00101 – Reject
- **M1** accepts exactly those strings in **A**
  where A = {w|w contains substring 11}
- So, **A is a language of M1** and **A = L(M1)**

Deterministic finite automata (DFA)

Non-deterministic finite automata (NFA)

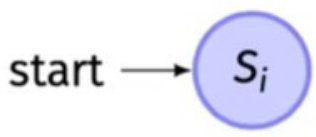
Implementation of Lexical Analysis

- Specify lexical structure using regular expressions.
- Tokenisation can be implemented using finite-state automata
  - Deterministic finite automata (DFA)
  - Non-deterministic finite automata (NFA)
- Implementation:
  - RegEx → NFA → DFA → Table-drive tokenisation

Finite State Machines

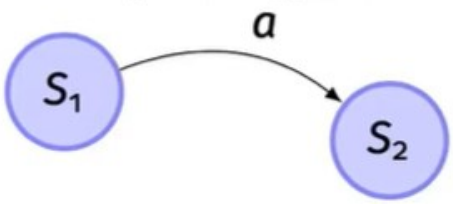
- A State

- Initial State



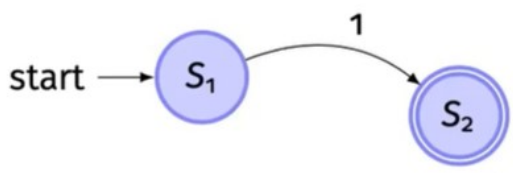
- Accepting State

- Transition: If in state *S*1, then on input *a* go to state *S*2.

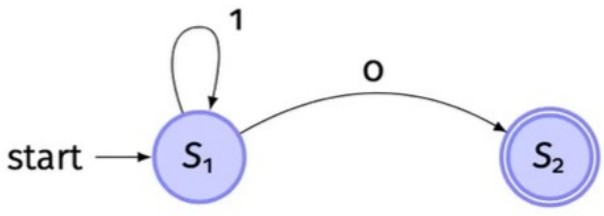


Simple Example

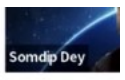
- A finite automata that accepts 1.



- A finite automata that accepts a sequence of 1s followed by a 0, over alphabet Σ = {1, 0}.



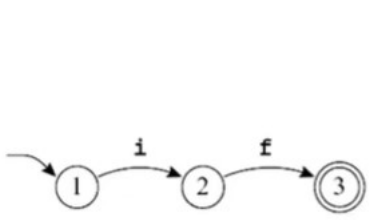
Simple Example



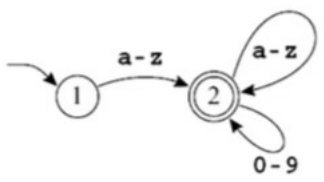
- **Keywords:** `if`
- **Identifiers:**

  `ID=[a-z][0-9a-z]*`
- **Integers:**

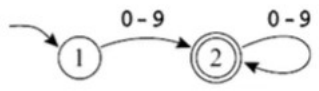
  `NUM=[0-9]+`



IF

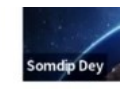


ID

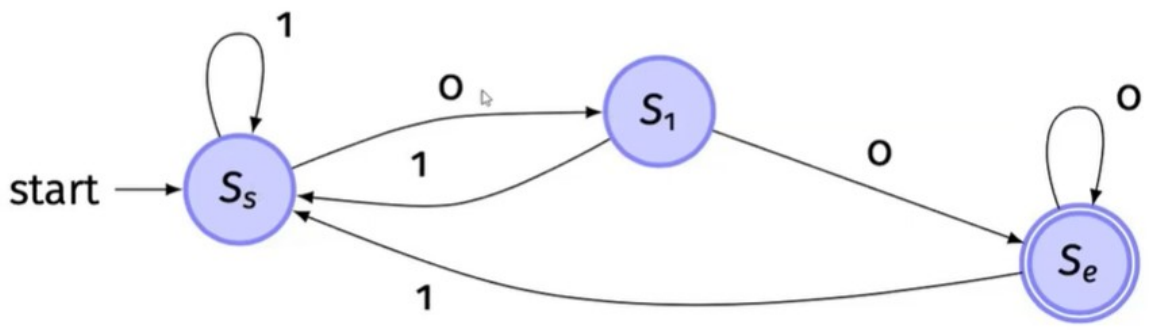


NUM

Another Example

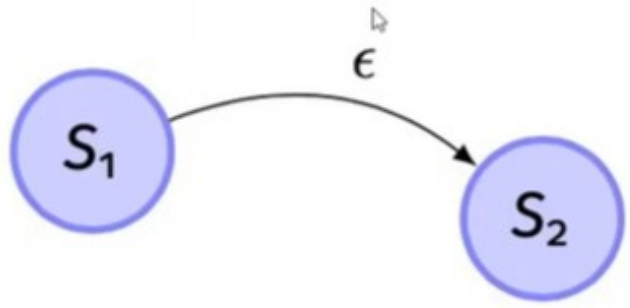


- Try to work out what this accepts

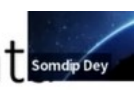


Epsilon Moves

- ε transitions
- Machine can change state with no input.



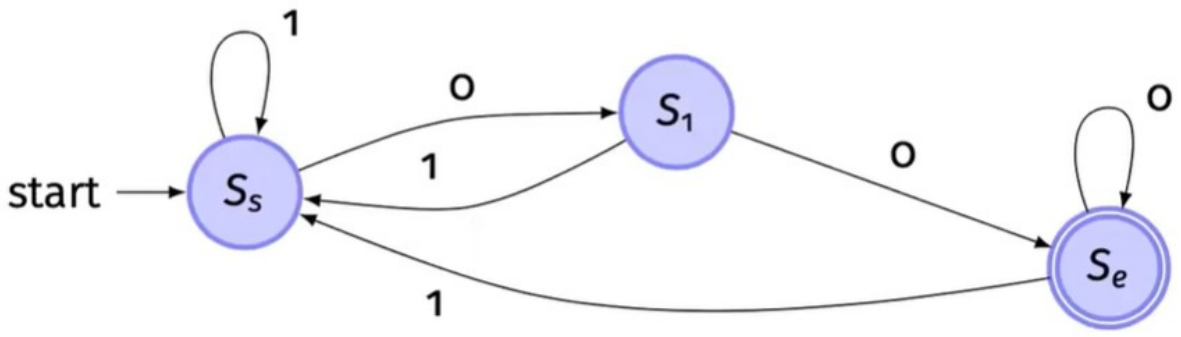
Deterministic and Non-deterministic Automat



- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ϵ-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ϵ-moves

Another Example

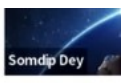
- Try to work out what this accepts



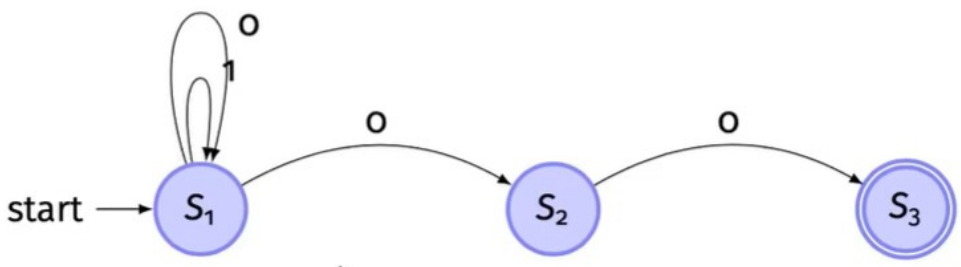
Execution of Finite Automata

- Deterministic Finite Automata (DFA)
  - Can only take one path through the state graph.
  - Transitions completely determined by the input.
  - Accepts when it reaches an "accept" state.
- Nondeterministic Finite Automata (NFA)
  - Can choose whether to make ϵ-moves.
  - Can choose which transition to take for a single input, when multiple transitions are available.
  - Accepts if there is some trace for that input which reaches an "accept" state.

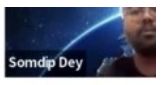
NFA Acceptance



- Consider the following with input 1 0 0.
- There is one execution path (trace) that reaches the accept state, so the string is accepted.
- But it requires us to consider all possible paths to determine this.

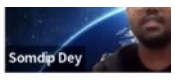


NFAs and DFAs



- Assuming NFAs and DFAs recognise the same set of languages.
- DFAs are easier to execute, as there no choices to consider, but can be (exponentially) larger than an equivalent NFA.
- NFAs can be more compact and simpler, as multiple choices expressed more simply but are more complex to execute directly.

NFA and DFA



- Notice in NFA on 00 input it can be still on S1 while on DFA it reaches the accepting state so what does equivalence means here?
- The definition of acceptance merely asserts that:
  - there must be some paths labelled by the input string in question leading from start state to an accepting state.
  - therefore the equivalence here means all strings that can be accepted by NFA (i.e. reach an accepting state) can also be reached by the equivalence DFA – and only in this very narrow sense these two automata are equivalent.

Regular Expression to NFAs

Generating a Lexical Analyser

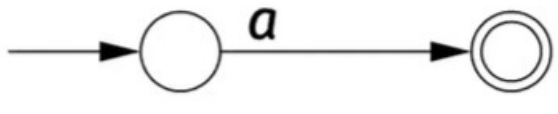
- Lexical Specification
- ↓
- Regular Expression
- ↓
- NFA
- ↓
- DFA
- ↓
- Table driven lexical analyser

Regex to NFA, basics

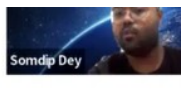
- General form of a regular expression *M*

- Empty actions ε

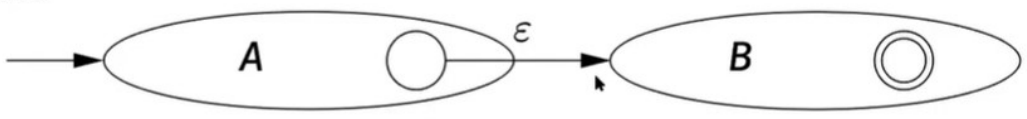
- Atomic regular expressions: input *a*



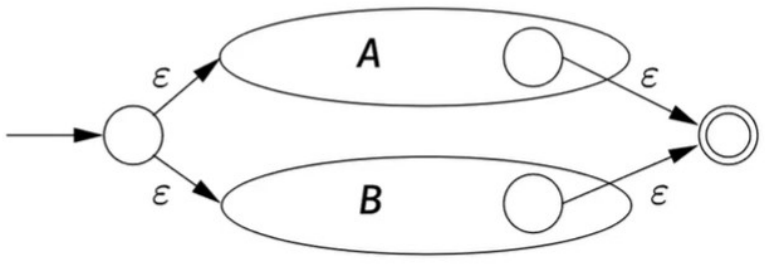
Regex to NFA, compound



- Concatenation *AB*

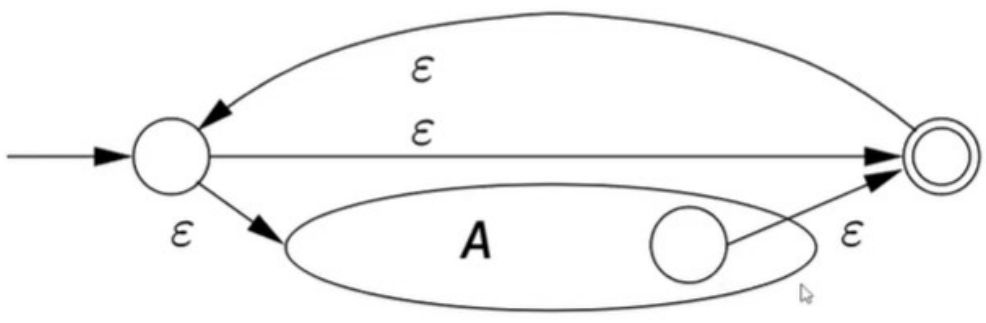


- Choice *A*|*B*

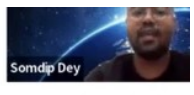


Regex to NFA, repetition

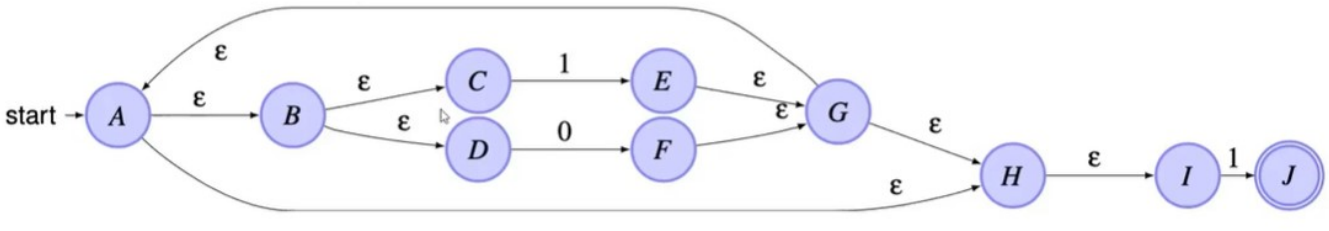
- Kleene star: *A**



Regex to NFA, example

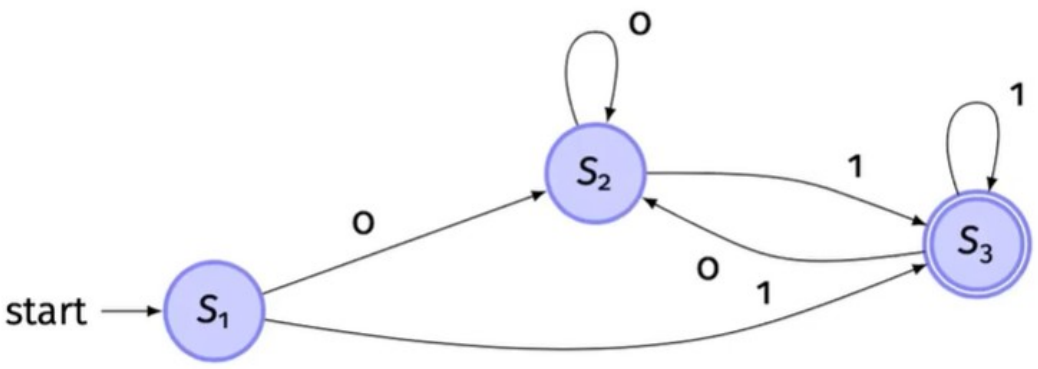


- Consider (1|0)*1
- As a NFA



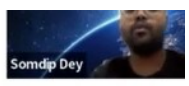
NFA to DFA, example

- Following from previous example, the NFA converted to a DFA:



where *S*1 = *ABCDHI*, *S*2 = *FGHIABCD*, *S*3 = *EJGHIABCD*.

NFA to DFA: background for an algorithm



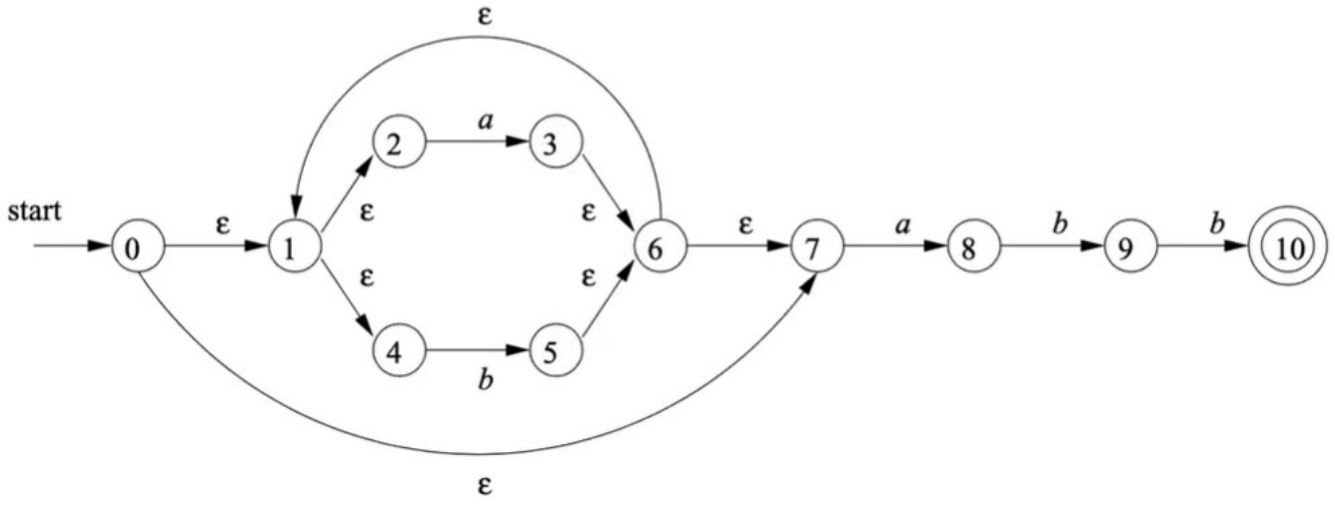
- In effect, we proceed by simulating the multiple-token (parallel) execution model of the NFA.
- Each state of DFA corresponds to a non-empty subset (collection) of the states of the NFA.
- The sets correspond to the multiple states in which the NFA may be after a given input sequence.
- Note that the number of states in the NFA may be much larger than in the DFA.
- We make a DFA from the NFA, such that each set of NFA states corresponds to one DFA state. Since the NFA has a finite number *n* of states, the DFA will also have a finite number (at most 2^n-1) of states .

NFA to DFA: sketch of an algorithm

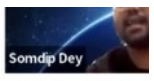
1. Start state of the DFA corresponds to the start state of the NFA and all the states reachable through ε moves.
2. We can add a transition $S_! \xrightarrow{"} S_\#$ to the DFA if $S_\#$ corresponds to the collection of states in the NFA reachable from any of the states in the NFA corresponding to $S_!$ on input symbol *a*, including with ε moves.
3. We continue until we have captured all permutations of states (and transitions) of the NFA in the DFA.

Example: Regular Expression to NFA

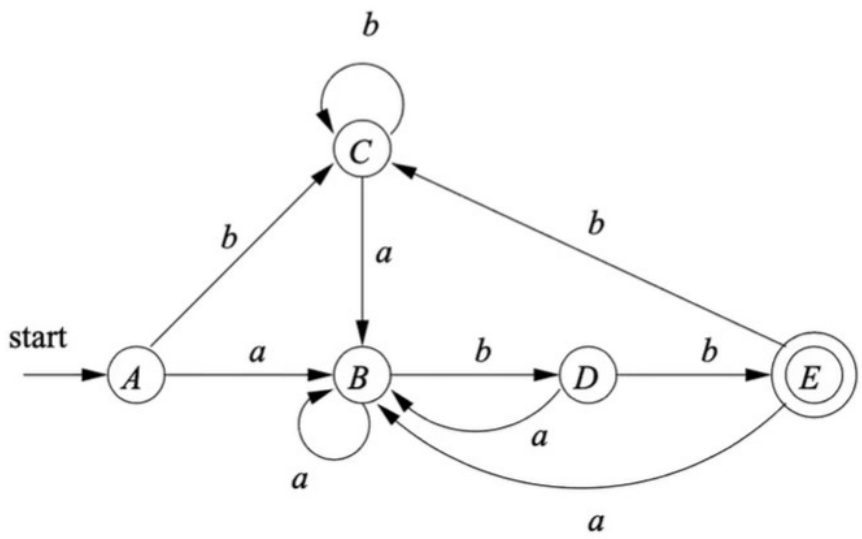
- Regular Expression: $(a|b)^*abb$
- Its NFA is:



Example: NFA to DFA



A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7},D={1,2,4,5,6,7,9},E={1,2,4,5,6,7,10}



Example: Regular Expression to NFA

- Regular Expression: $(a|b)^*abb$
- Its NFA is:

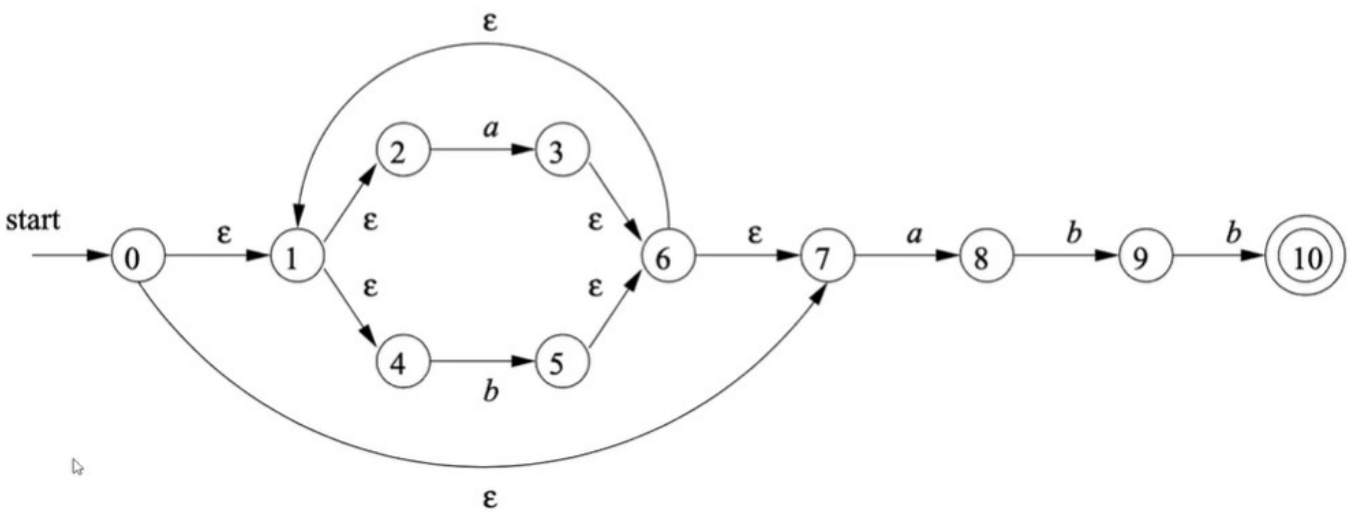
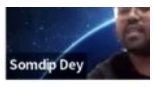


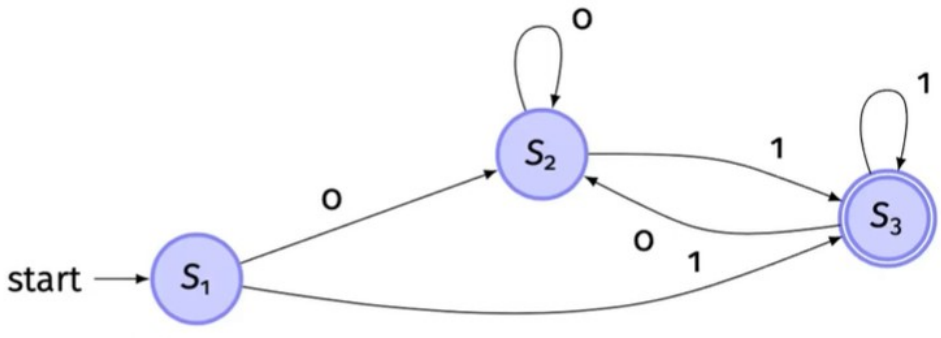
Table driven tokenisation

- Once we have the DFA, it is easy to translate this into a table-driven algorithm for recognising tokens.
- Each row corresponds to a state.
- Each column corresponds to an input.
- Each cell tells us which state to move to for a given input in a given state.

Example Table



- Consider the DFA:



- This can be tabulated as follows:

	0	1
S_1	S_2	S_3
S_2	S_2	S_3
S_3	S_2	S_3

Other considerations

- To save space (at the expense of speed), it may not always be appropriate to produce the full DFA.
- The lexical analysis needs to indicate the tokens found, and any "semantic" information.
- These steps, and optimisations, are automated by tools.

Grammars and Parsing - part 1

Grammars

Background

- While looking into tokenisation we have seen the use of formal languages, automata, expressivity v. efficiency and implement ability.
- All of these are foundational issues in computer science.
- For tokenisation, we only needed regular grammars.
- For parsing we will need something with greater power.

Languages and Automata

- Different constraints can be applied to the nature of the rules of a grammar.
- These constraints may restrict both the languages that can be recognised, and also the power of the computational device required to parse using the grammar.

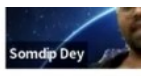
Regular Languages

- Regular Grammars can be implemented using finite state systems.
- Keep in mind that not all useful languages can be expressed using Regular Grammars.
- Finite state systems (and the rules of regular grammars), have no memory. They cannot "count" the number of times a particular symbol has appeared, or a state has been entered.

More Needed

- Is this a problem?
- Consider parenthesis matching: $((8 \times 2) + (((2 + 3) \times 2) \div 10))$
- For the parenthesis to match, the rules of grammar need to be able to express the requirement that
  - by the end of the expression, the number of "("s equals the number of ")"s, and
  - at any point within the expression we never encounter more ")"s than the number of "("s we have seen (going left-to-right).
- In effect, to parse a grammar that is sufficiently expressive to have matching parenthesis, we need some form of storage.
- In general, to parse such grammars, we would need a stack in addition to a finite state automata.
- A finite state machine can "simulate" constrained parenthesis matching, but the solution is not general, and is not easy to express or efficient to implement.

Grammar Formal Definition

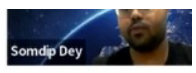


- In formal language theory, a grammar defines how to form strings from a language's alphabet that are valid according to the language's syntax.
- A grammar is a tuple $\{N, T, P, S\}$
  - N is a finite set of non-terminals
  - T is a finite set of terminals, s.t. $N \cap T = \emptyset$
  - P is a finite set of production rules in the form $\alpha \rightarrow \beta$ where $\alpha \in (N \cup T)^+$ and $\beta \in (N \cup T)^*$
  - $S \in N$ the start symbol

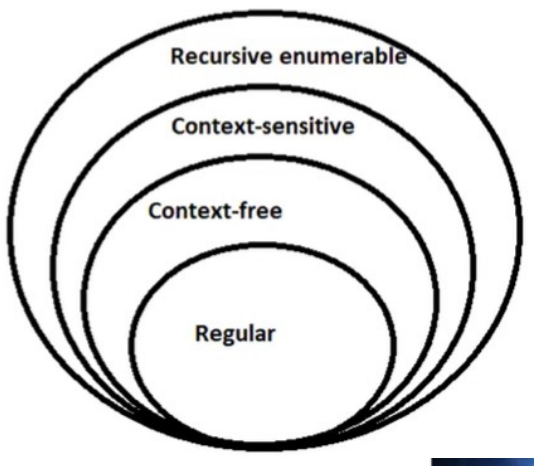
Chomsky's Hierarchy

- The **Chomsky hierarchy** is a containment hierarchy of classes of formal grammars.
- It is a characterisation of languages where the corresponding grammars are classified according to
  - the kinds of rules permitted
  - the kinds of sentence structures that can(not) be recognised
  - the kinds of automata that can implement parsers for that class of languages.

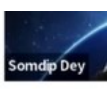
Chomsky hierarchy



- Chomsky's four types of grammars and the class of language it generates as as follows:
- Type 0 Unrestricted grammar.
- Type 1 Context sensitive grammar.
- Type 2 Context free grammar.
- Type 3 Regular Grammar.



Regular Grammars

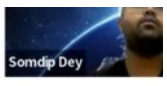


- A **regular grammar** is a formal grammar that is right-regular or left-regular. Every regular grammar describes a regular language.
- **Type** 3
  - **Rules:** A → x B and A → x where x is a terminal category
  - **Computational Power:** Finite State Automata
  - **Characteristic String:** a^*b^*

Context Free Grammars

- A **context-free grammar** (CFG) is a formal grammar whose production rules are of the form A → α with A being a single nonterminal symbol, and α a string of terminals and/or non-terminals (where α can be empty).
- A formal grammar is "context free" if its production rules can be applied regardless of the context of a nonterminal.
- **Type 2**
  - **Rules:** A → B C D ⋯
  - **Computational Power** Push Down Stack Automata
  - **Characteristic String:** $a^n b^n$ (nested dependencies)
  - The Left Hand Side Symbol is a non-terminal

Context Sensitive Grammars



- A **context-sensitive grammar** (CSG) is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols.
- Context-sensitive grammars are more general than context-free grammars, in the sense that there are languages that can be described by CSG but not by context-free grammars.
- **Type 1**
  - **Rules** A B C → A D C
  - **Computational Power** Linear Bound Automata
  - **Characteristic String** $a^n b^n c^n$ (cross-serial dependencies)
  - The number of left hand side symbols ≤ the number of right hand side symbols

Recursively enumerable Grammars

- Recursively enumerable or Turing acceptable language will enumerate all valid strings of the language.
- Recursively enumerable languages are known as type-0 languages in the Chomsky hierarchy of formal languages. All regular, context-free, context-sensitive and recursive languages are recursively enumerable.
- **Type 0**
  - **Rules** *anything* → *anything*
  - **Computational Power** General Turing Machine
  - **Characteristic String** (Any dependencies)

Language and Automata Hierarchy

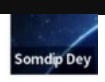
- Increasingly powerful automata can parse increasingly complex grammar families, from Type 3 to Type 0.
- Unfortunately, more powerful formalisms have some unpleasant properties that are related to the halting problem.
- The halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.
- Fortunately, programming languages (and most human language) are expressible in context free grammars (CFGs).
- Programming languages in general require something less than the full potential of CFG, while human languages sometimes require slightly more expressiveness than CFG.
- Even so, computational complexity of parsing may be high, and some obvious questions are undecidable (e.g. the equivalence relationship).

Other relevant classes

- Chomsky's hierarchy is not the only classification, but a pertinent one for theoretical computer science.
- When it comes to programming languages, other relevant classes include *LL(k)* and *LR(k)*, for example (where k = 0, 1, ⋯) and k is the number of lookaheads.
- When it comes to linguistics, other relevant classes include weakly context sensitive grammars.

Context Free Grammar

Regular Grammars



- $G = (N,T,P,S)$ is a regular grammar if:
  - all productions are the form
    - $A \rightarrow a$

      where $A \in N, a \in T$ or
    - $A \rightarrow aB$

      where $A,B \in N$ and $a \in T$
  - ϵ-production of the form $A \rightarrow \epsilon$, then *A* does not appear as a substring on right-hand side of any other production in *P*

Context Free Grammars

- $G = (N, T, P, S)$ is a Context Free Grammar if production rules in *P* are of the form:

  $A \rightarrow \alpha\gamma\beta$

  where $A \in N$, $\alpha,\beta \in (N \cup T)^*$ and $\gamma \in (N \cup T)^+$

Context Free Grammars

- The recursive syntactic structure of programs can be defined by a context free grammar (CFG).
- CFGs have
  - **terminal symbols** *T*, the tokens assigned by the lexical analyser.
  - **non-terminal symbols** *N*, representing "phrases" of the language.
  - **start symbol** a distinguished non-terminal corresponding to the category of the largest structure we wish to find.
  - **rules** of the form

    $X \rightarrow Y_1 Y_2 \cdots Y_n$, where $X \in N$, $Y_j \in N \cup T$
  - These rules tell us which categories we need, and the order in which they occur for us to have category *X*.
- The application of CFG rules can be seen to generate tree structures.

Example: Arithmetic Expressions

$E \rightarrow \text{VAR}(id)$
$E \rightarrow \text{NUM}(val)$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow -E$
$E \rightarrow (E)$

Example: Boolean Expressions

$B \rightarrow E = E$
$B \rightarrow E < E$
$B \rightarrow !B$
$B \rightarrow (B)$
$B \rightarrow B$ and B
$B \rightarrow B$ or B

Example: Statements

$S \rightarrow \text{VAR}(int) := E$
$S \rightarrow$ if B then S
$S \rightarrow$ if B then S else S
$S \rightarrow$ while B S
$S \rightarrow$ begin Ss end

$Ss \rightarrow S$
$Ss \rightarrow S; S$

Backus-Naur Form (BNF)

Backus-Naur Form (BNF)

- Backus–Naur form or Backus normal form is a metasyntax notation for context-free grammars, often used to describe the syntax of programming languages.
- Alternative presentation of rules.
  - Essentially uses := (or =) in place of →
  - and | to indicate a choice of right sides.
- An example *E*:=VAR(*id*) | NUM(*val*) | *E* + *E* | *E* ∗ *E* | −*E* | (*E*)
- It is compact, and can be written without using special symbols.

Example BNF Grammar

E:= VAR(*id*) | NUM(*val*) | *E*+*E* | *E*∗*E* | −*E* | (*E*)
B:= *E*=*E* | *E*<*E* | !*B* | (*B*) | *B* and *B* | *B* or *B*
S:= VAR(*int*) := *E* | if *B* then *S* | if *B* then *S* else *S*
| while *B S* | begin *Ss* end
Ss:= *S* | *Ss*; *Ss*

Extended Backus-Naur Form (EBNF)

- An extension to BNF proposed by Niklaus Wirth.
- One version (ISO/IEC 14977):

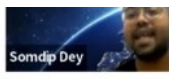
definition	=
concatenation	,
termination	;
alternation	|
option	[...]
repetition	{...}
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	?...?
exception	-

Variations on (E)BNF

- Not all sources use standard BNF or EBNF notation.
- One example is the notation that defines the category.
- Common symbols include :=, =, and →.

Parsing

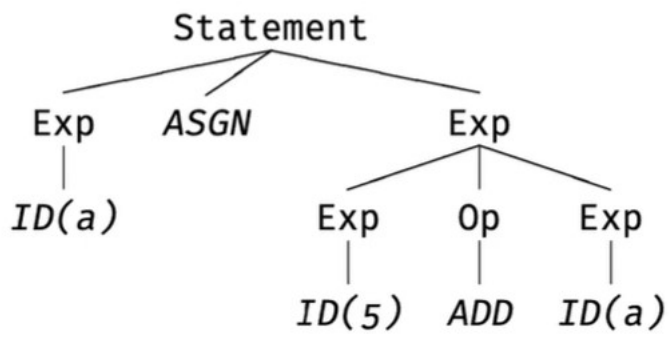
Parsing



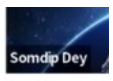
- Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols in programming languages.
- For our compiler, we need a parser that has
  - **input** lexemes from the tokenizer
  - **output** parse tree of the program
- In practice the parser may not actually produce the parse tree itself as output, but instead some other information that was generated in the process corresponding to construction of the parse tree, such as an abstract syntax tree, or a semantic interpretation.

Parsing Example

- Assume the original input was "a := 5 + a"
- And that the tokeniser produced "*ID(a), ASGN, ID(5), ADD, ID(a)*"
- Then (given an appropriate grammar) we might expect to obtain the parse tree:



Comparison



- Tokeniser takes a string of characters and produces a string of tokens.
  - Its role is to recognise and assign valid tokens.
  - We have a language for stating rules about valid tokens, and ways of using those rules to recognise valid tokens (and assign appropriate values where relevant).
- Parser takes a string of tokens and produces a parse tree.
  - The role of the parser is to recognise valid programs, and allow a "meaning" to be assigned.
  - Parsers may give helpful information in the event a program is not valid.
  - We need a language for stating rules about valid programs, and an algorithm that uses those rules to recognise valid programs and assign an appropriate structure.
- Both tokenisers and parsers may give helpful information in the event a string or program is not valid.

Derivations

- The rule $X \rightarrow Y_1\ Y_2 \cdots Y_n$ can be read as an instruction.
  - If we have *X*, it can be replaced by $Y_1\ Y_2 \cdots Y_n$ (top down)
  - If we have $Y_1\ Y_2 \cdots Y_n$, it can be replaced by *X* (bottom up)
- Can apply the rules (top down):
  1. Begin with the start symbol.
  2. Replace any non-terminals *X* by the right side of a rule of the form $X \rightarrow Y_1\ Y_2 \cdots Y_n$
  3. Continue until there are no non-terminals left.

Derivations (cont)

- From $X_1 X_2 \cdots X_i \cdots X_n$, we can obtain
  $X_1 X_2 \cdots Y_1 Y_2 \cdots Y_m \cdots X_n$
  If there is a rule $X_i \rightarrow Y_1 Y_2 \cdots Y_m$
- We can write
  $X_1 X_2 \cdots X_n \Longrightarrow^* Y_1 Y_2 \cdots Y_m$
  to say the latter can be derived from the former in zero or more steps.

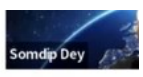
Languages generated by CFGs

- Assume we have a context free grammar *G*.
- The language *L*(*G*) generated by *G* will be all the possible sequences of terminals $a_1 \cdots a_n$ which can be derived from the start symbol *S* using the rules.
- $L(G) = \{a_1 \cdots a_n \mid S \Longrightarrow^* a_1 \cdots a_n\}$ where $\Longrightarrow^*$ is defined by *G*.

Derivation Examples

- Given a grammar: $E \rightarrow E{+}E \mid E{*}E \mid -E \mid (E) \mid$ id
- For the production $E \rightarrow -E$, the replacement of a single *E* by -*E* will be described by
  $E \Longrightarrow -E$
  which is read as "*E* derives -*E*".
- The production $E \rightarrow (E)$ can be applied to replace any instance of *E* in any string of grammar symbols by (*E*).
  $E{*}E \Longrightarrow (E){*}E$ *or* $E{*}E \Longrightarrow (E){*}(E)$
- Repeatedly apply productions in any order to get a sequence of replacements:
  $E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow -(\text{id})$

Derivation Examples



- **Leftmost** derivations: the string -(id+id) is a sentence of the grammar because there is a derivation:
  $E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow -(E + E) \Longrightarrow -(\text{id} + E) \Longrightarrow -(\text{id}+\text{id})$
- **Rightmost** derivations: the string -(id+id) is a sentence of the grammar because there is a derivation:
  $E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow -(E + E) \Longrightarrow -(E + id) \Longrightarrow -(\text{id}+\text{id})$

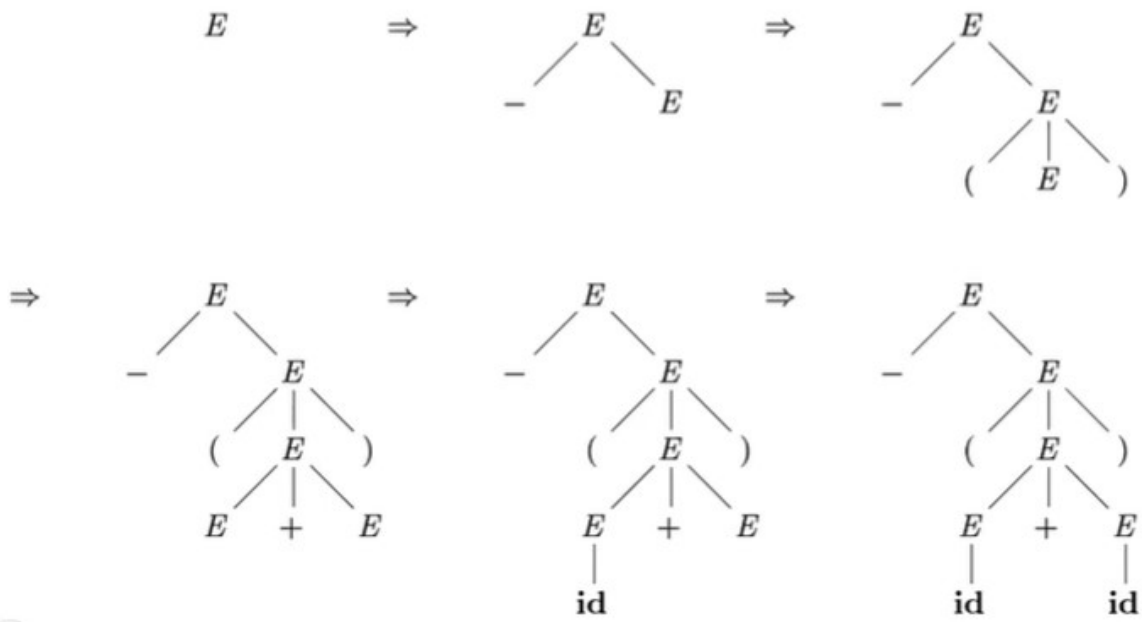
Parse Trees and Derivations

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals
- Each interior node of a parse tree represents the application of a production.
- The interior node is labelled with the non-terminal A in the head of the production.
- The children of the node are labelled from left to right by the symbols in the body of the production by which this A was replaced during the derivation.
- The leaves of a parse tree are labelled by non-terminals or terminals and read from left to right.

Parse Trees

- The parse tree for –(id+id) with leftmost derivations:

$$E \Longrightarrow -E \Longrightarrow -(E) \Longrightarrow -(E + E) \Longrightarrow -(\text{id} + E) \Longrightarrow -(\text{id}+\text{id})$$

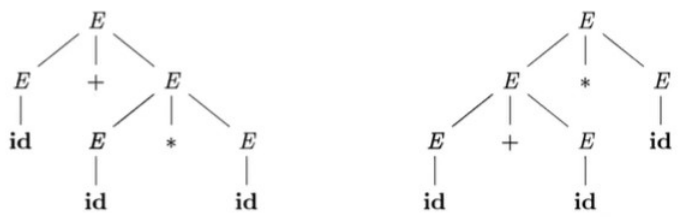


Grammar Ambiguous

- A grammar: $E \rightarrow E{+}E \mid E{*}E \mid -E \mid (E) \mid \text{id}$
- It will produce two leftmost derivations for the sentence

  id+id*id
- The parse tree on the left reflects the commonly assumed precedence of ”+” and “∗” while the tree on the right does not.
- To add precedence: force the parser to recognise high-precedence subexpressions first.

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$



Parsing and Derivations

- There are many different parsing algorithms.
- They can be characterised in different ways.
- One dimension of classification is the direction in which the rules are applied.
  - Top-down Rule driven. Use rules to expand from left to right.
  - Bottom-up Data driven. Use rules to compress from right to left.
- In the case of CFGs, the derivation can be recorded structurally as a tree.
- You should know how to apply the rules manually.

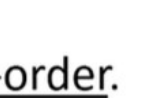
Error Recovery

- A syntax error occurs when the string of input tokens is not a sentence in the language.
- Error recovery is a way of finding some sentence similar to that string of tokens.
- This can proceed by deleting, replacing, or inserting tokens.
- It's a bit dangerous to do error recovery by insertion, because if the error cascades to produce another error, the process might loop infinitely.
- Error recovery by deletion is safer, because the loop must eventually terminate when end-of-file is reached.

Grammars and Parsing - part 2

Quick recap

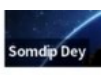
Quick recap on Parser



- Parser is a compiler or interpreter component that takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.
- Parser takes an input string and it will tell whether the input string is in accordance with the grammar production or not, which means it will either accept the string or will reject it with certain error.
- Parser can be of two types: Top-Down Parsing & Bottom-Up Parsing.

Top-Down Parsing

Top-Down Parsing



- **Top-down parsing** is a **parsing** strategy where one first looks at the highest level of the **parse** tree and works **down** the **parse** tree by using the rewriting rules of a formal grammar.

Steps:

- Construct the top node of the tree and then the rest in pre-order. (depth-first)
- Pick a production and try to match the input; if you fail, backtrack.
- Essentially, we try to find a **leftmost** derivation for the input string (which we scan left-to-right).
- Some grammars are backtrack-free (predictive parsing).

Top-Down Parsing, recursive descent

- **Recursive descent** is a **top-down parsing** technique that constructs the **parse** tree from the **top** and the input is read from left to right. It uses procedures for every terminal and non-terminal entity.

Steps:

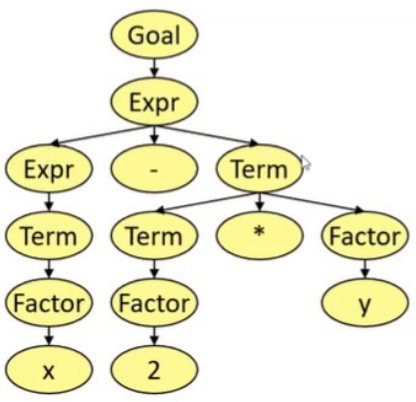
- Construct the root with the starting symbol of the grammar.
- Repeat until the fringe of the parse tree matches the input string:
  - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
  - When a terminal symbol is added to the fringe.
  - Find the next node to be expanded.

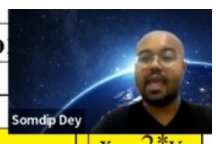
Top-Down Parsing Example

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Top-Down Parsing Example

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!





Rule	Sentential Fo	
-	*Goal*	
1	*Expr*	
2	*Expr* + *Term*	| x – 2*y
4	*Term* + *Term*	| x – 2*y
7	*Factor* + *Term*	| x – 2*y
9	*id* + *Term*	| x – 2*y
Fail	*id* + *Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr* – *Term*	| x – 2*y
4	*Term* – *Term*	| x – 2*y
7	*Factor* – *Term*	| x – 2*y
9	*id* – *Term*	| x – 2*y
Match	*id* – *Term*	x – | 2*y
7	*id* – *Factor*	x – | 2*y
9	*id* – *num*	x – | 2*y
Fail	*id* – *num*	x – 2 | *y
Back	*id* – *Term*	x – | 2*y
5	*id* – *Term* * *Factor*	x – | 2*y
7	*id* – *Factor* * *Factor*	x – | 2*y
8	*id* – *num* * *Factor*	x – | 2*y
match	*id* – *num* * *Factor*	x – 2* | y
9	*id* – *num* * *id*	x – 2* | y
match	*id* – *num* * *id*	x – 2*y |

Left Recursive

- A grammar is **left-recursive** if it has a non-terminal symbol *A*, such that there is a derivation $A \rightarrow A\ \alpha \mid \beta$, where A is nonterminal & α, β are tokens.
- It can also arise indirectly

  $A \rightarrow B\alpha \mid C$

  $B \rightarrow A\beta \mid D$
- Note: A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Left Recursion - Solutions

- Consider the following grammar:

  $A \rightarrow A\alpha \mid \beta$
- We can remove the left recursion in the following way and obtaining an equivalent grammar (by converting left recursion to right recursion):

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$
- For a second example consider:

  $A \rightarrow A\beta\alpha \mid A\alpha \mid \alpha$
- This can be transformed as:

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta\alpha A' \mid \varepsilon$

  $A' \rightarrow \alpha A' \mid \varepsilon$

Left Recursion — solutions

- Avoid left-recursive rules
  - There are general solutions to this, replacing left recursion with right recursion.
  - For immediate left-recursion, we can replace rules of the form

    $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_n$

  by

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

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

    where α cannot include ε, and β excludes *A*-initial sequences.
  - This can be generalised to indirect left-recursive rules.
- Avoid (naive) recursive descent parsing.

Recap

- Top-down parsing
  - recursive with backtracking (not often used in practice)
  - recursive predictive (without backtracking)
- We can produce a top-down parser. If it picks the wrong production rule it has to backtrack.
- Recursive predictive parsing works only on grammars where the *first terminal symbol* of each subexpression provides enough information to choose which production to use.
- How about look ahead in input and use context to pick correctly?
- How much lookahead is needed?
  - In general, an arbitrarily large amount.
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

LL(k) Grammars

- LL(k): left-to-right parse, leftmost derivation, k-token lookahead.
- In formal languages, an LL grammar is a context-free grammar that can be parsed by an LL parser, which parses the input from Left to right, and constructs a Leftmost derivation of the sentence.
- To resolve all potential conflicts, the parsing algorithm has to look ahead a certain number of characters (k).
- This gives rise to additional hierarchical class of grammars known as $LL(k)$ grammars that recognise a subset of the CFG languages.
- There are all kinds of questions that can be considered here, such as whether the strings of a language generated by a given $LL(k)$ grammar can be recognised by an $LL(k')$ grammar, where $k' < k$.

LR Parsing

- The weakness of LL(k) parsing techniques is that they must *predict* which production to use, having seen only the first k tokens of the right-hand side.
- A more powerful technique, LR(k) parsing, is able to postpone the decision until it has seen input tokens corresponding to the entire right-hand side of the production.
- LR(k) stands for *left-to-right parse, rightmost-derivation, k-token lookahead.*

Bottom-Up Parsing

- A **bottom-up parse** discovers and processes that tree starting from the **bottom** left end, and incrementally works its way **upwards** and rightwards.

Steps:

- Construct the tree for an input string, beginning at the leaves and working up towards the top (root).
- Bottom-up parsing, using left-to-right scan of the input, tries to construct a **rightmost** derivation in reverse.
- Note: Handle a large class of grammars.

Bottom-Up Parsing

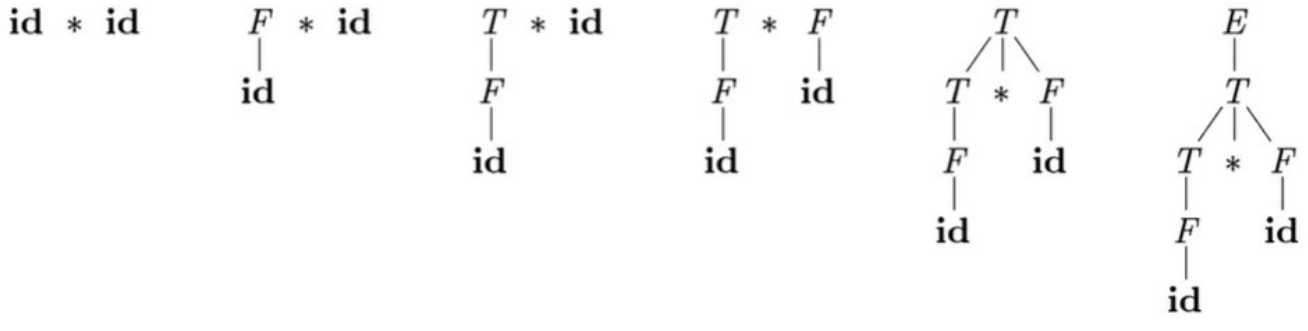
- Expression grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid id$

- For token stream id $*$ id



Bottom-Up Parsing: shift-reduce parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- We use $ to mark the bottom of the stack and also the right input
- Initially, the stack is empty, and the string w is on the input

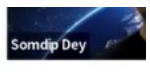
STACK	INPUT
$	w $

Bottom-Up Parsing: shift-reduce parsing

- For token stream id $*$ id

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Bottom-Up Parsing: shift-reduce parsing



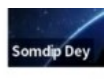
- The parser **shifts** zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then **reduces** β to the LHS of the appropriate production.
- The parser repeats this cycle until the stack contains the start symbol and the input is empty. Then the sentence is valid

LR(k) Grammars

- This is known as *LR*(*k*) parsing. The *k* relates to the number of symbols that have to be considered when generating the table.
- As with *LL*(*k*) definable languages, there are questions as to whether the language generated by a given *LR*(*k*) can be recognised with an *LR*(*k*′) grammar where *k*′ < *k*.
- LR(1) grammars are widely used to construct (automatically) efficient and flexible parsers

Grammar Ambiguity

Ambiguous Grammar



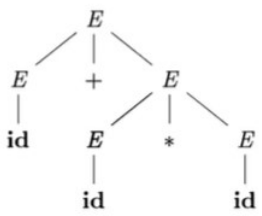
- An **ambiguous grammar** is a context-free **grammar** for which there exists a string that can have more than one leftmost derivation or parse tree, while an unambiguous **grammar** is a context-free **grammar** for which every valid string has a unique leftmost derivation or parse tree.

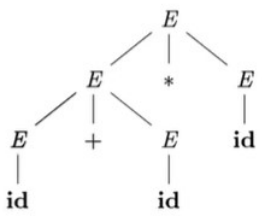
Grammar Ambiguous

- A grammar: *E* → *E*+*E* | *E*∗*E* | -*E* | (*E*) | id
- It will produce two leftmost derivations for the sentence

  id+id*id
- The parse tree on the left reflects the commonly assumed precedence of ”+” and “∗” while the tree on the right does not.
- To add precedence: force the parser to recognise high-precedence subexpressions first.

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$





Ambiguity and Top-Down parsing

- With top-down parsing, a different choice of rules can generate the sentence.
- We could generate all possible parses by forcing backtracking even when we recognise a sentence.
- The syntactic structures corresponding to all permutations of the application of the rules that lead to the sentence being recognised represent all the possible parses.

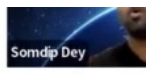
Ambiguity and Bottom-Up Parsing

- With bottom-up shift reduce parsing, different choices of reducing and shifting yield different parses.
- When attempting to parse programming languages with an LR(k) table-driven grammar, this is sometimes called shift-reduce conflict, and reduce-reduce conflict.

Views on Ambiguity

- Ambiguity is considered to be an intrinsic feature of natural language, at all levels of analysis.
- Parsers for natural language may be designed to be efficient at finding all possible analyses of a sentence.

Views on Ambiguity



- With designed languages, such as programming languages, ambiguity is usually considered to be a bad thing that should be avoided.
- Parsers for programming languages may be designed to be efficient on the assumption that the language is not ambiguous.
- This assumption is often relied upon in LL(k) and LR(k) parsing.
- Hence “shift-reduce, and reduce-reduce, conflict” needs to be resolved.

Shift/Reduce & Reduce/reduce conflicts

- **Shift/reduce conflicts**: the parser cannot decide whether to shift or to reduce.

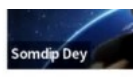
  Example: the dangling-else grammar; usually due to ambiguous grammars.

  Solution: a) modify the grammar; b) resolve in favour of a shift.
- **Reduce/reduce conflicts**: the parser cannot decide which of several reductions to make.

  Example: `id(id,id)`; reduction is dependent on whether the first `id` refers to array or function.

  May be difficult to tackle.

Avoiding Ambiguity



- Re-expressing the grammar
  - Sometimes the grammar can be re-written to avoid unintended ambiguity.
  - This is not always desirable, as the resulting grammar can be hard to read.
- Operator/rule precedence
  - Resolve the ambiguity by imposing a precedence on the operators.
  - This can be implemented in LR(k) parsing by having the table determine the next operation in those cases where a choice of operations would otherwise be available.

Ambiguity: dangling else

- What is the meaning of the following?

```
If B1 then if B2 then S1 else S2
```

Ambiguity: dangling else

- Assume the grammar
  - if-statement → "if" boolean, statement
    → "if" boolean, statement, "else", *statement*
  - Statement → simple-statement
    → if-statement
    → loop-statement
  - loop-statement → "while" boolean, statement
- Now "`if(A)if(B)statementA; else statementB;`" is ambiguous between:
  - `if (A) { if (B) statementA; else statementB; }`
  - `if (A) { if (B) statementA; } else statementB;`

Resolving dangling else

- The dangling else can be resolved using precedence rules, or similar “tricks”, to favour **the first interpretation**.
- An **alternative** is to rewrite the grammar:

statement	→open-statement →closed-statement
open-statement	→“if” boolean, statement →“if” boolean, closed-statement, “else”, open-statement →“while” boolean, open-statement
closed-statement	→simple-statement →“if” boolean, closed-statement, “else”, closed-statement →while boolean, closed-statement

Resolving dangling else

How does this work?

- Note that “open” statements have at least one “if” not paired with “else”.
- And if a “closed” statement includes “if” statements, they must be paired with an “else”.
- So given the rules for “if. . . else”, a conditional statement with an “else” can only contain another conditional in the “then” branch when that conditional statement comes with its own “else” clause.
- Now, the only way to parse the dangling else is to associate it with the final “if” statement.
- Note that the parse tree differs from that of the original (*cf. “weak generative capacity”*).

Abstract Syntax Trees

Concrete Syntax Trees

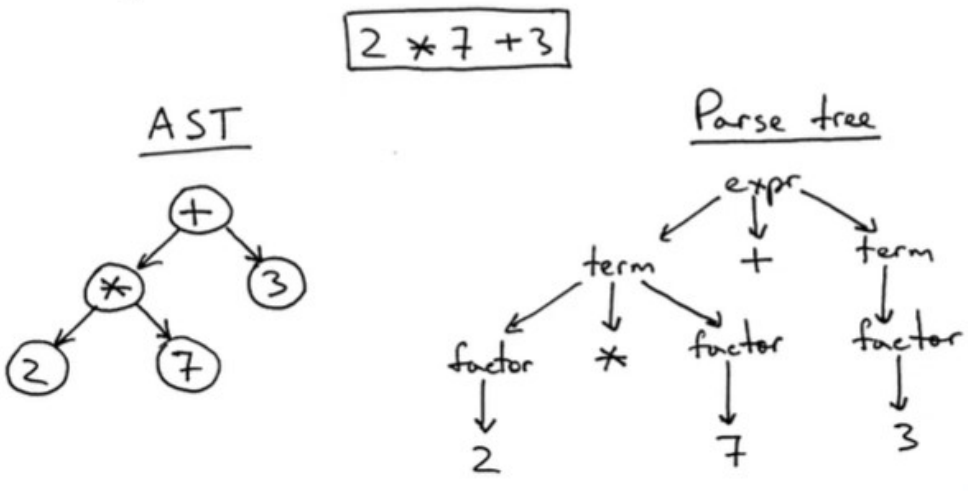
- A parse-tree (sometimes called a *concrete syntax tree*) is a tree that represents the syntactic structure of a language construct according to our grammar definition.
- It basically shows how your parser recognized the language construct.
- In concrete parse trees, many of the punctuation tokens are redundant and convey no information - they are useful in the input string.
- Elimination of left recursion or elimination of ambiguity involves the introduction of extra non-terminals and extra grammar productions. These details should be confined to the parsing phase.

Abstract Syntax Trees

- An *abstract syntax* makes a clean interface between the parser and the later phases of a compiler.
- The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.
- AST is one of the intermediate representation (IR)
- We can abstract away from this concrete syntax by
  - deleting intermediate non-branching categories and
  - pushing infix operators up to the nearest branching parent node.

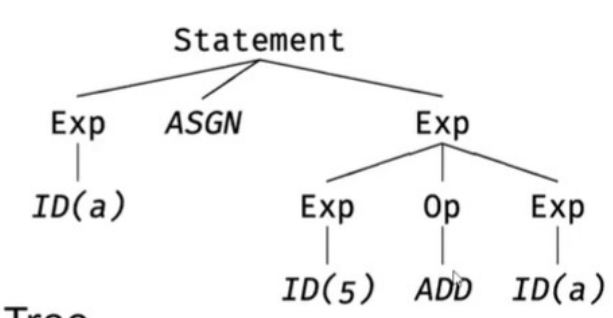
AST Example

- The AST and the parse tree for the expression 2 * 7 + 3

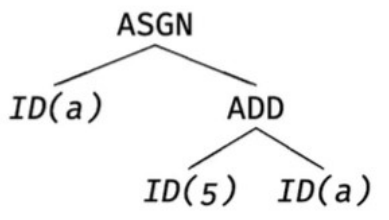


Concrete vs. Abstract Syntax

- A Concrete Syntax Tree



- The corresponding Abstract Syntax Tree

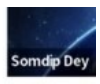


Concrete vs. Abstract Syntax

- ASTs uses operators/operations as root and interior nodes and it uses operands as their children.
- ASTs do not use interior nodes to represent a grammar rule, unlike the parse tree does.
- ASTs don't represent every detail from the real syntax (that's why they're called *abstract*) - no rule nodes and no parentheses, for example.
- ASTs are dense compared to a parse tree for the same language construct.

Implementation Code Examples

Balanced parentheses



- Consider the following simple language (no left recursion):

$$B \rightarrow \varepsilon$$
$$B \rightarrow BB$$

- Example of this language ε, (), (()), ()()(())
- Next we show how to write a recursive descent parser for this grammar
- Notice in the following code we separated each token in the input stream with space-character to avoid tokenisation – just concentrating on the parser.

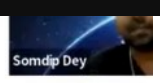
A simple expression language

- No left recursion

$$E \rightarrow TE'$$
$$E' \rightarrow \varepsilon \mid +TE'$$
$$T \rightarrow FT'$$
$$T' \rightarrow \varepsilon \mid * FT'$$
$$F \rightarrow (E) \mid num$$

Introduction to Semantics - part 1

Overview



- We have considered tokenization and parsing
- Both are steps to producing a compiled, and both can be used to detect errors
- The next step is “static semantic analysis”
- The aim here is to add additional contextually dependent information to the parse tree.
- This is required to produce appropriate compiler output and detect certain kinds of “semantic” error.

Context Dependence

- Programs that are correct with respect to the language’s lexical and context-free syntactic rules may still contain other syntactic errors
- Lexical analysis (tokenization) and conventional context-free syntax analysis (parsing) are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.
- When speaking of compilers, those aspects of interpretation that require an analysis of execution context are often known as semantic analysis.
- (This is not a conventional definition of semantics, even in computer science. Some texts also claim such analysis requires something more powerful than CFG, but this is a bit debatable.)

Incorrect Programs (Examples)

- Parsing does not check
  Variable names: int a;. . . a := 1; is the “same” as
  int b;. . . a := 1; to the parser.
- Existence of declarations a := 1; is as valid as int a;. . . a := 1;
- Types of variables and expressions int a;. . . a := 1.0; is as syntactically well-formed as int a;. . . a := 1;

- Henceforth, these can be checked using “semantic analysis” in your language.

Goals of Semantic Analysis

- Semantic analysis ensures that the program satisfies a set of additional rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be declared before being used
  - A variable should not be declared multiple times in the same scope
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The condition of an if-statement must have type Boolean
- Some of the categories of semantic rules:
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type Information

- Type information classifies a program's constructs (e.g., variables, statements, expressions, functions) into categories, and imposes rules on their use (in terms of those categories) with the goal of avoiding runtime errors

Category	*Example*	*Type*
variables:	`int a;`	integer location
expressions:	`(a+1) == 2`	Boolean
statements:	`a = 1.0;`	void
functions:	`int pow(int n, int m)`	int × int → int

- You need Type Checking to validate the rules

What is Type Checking?

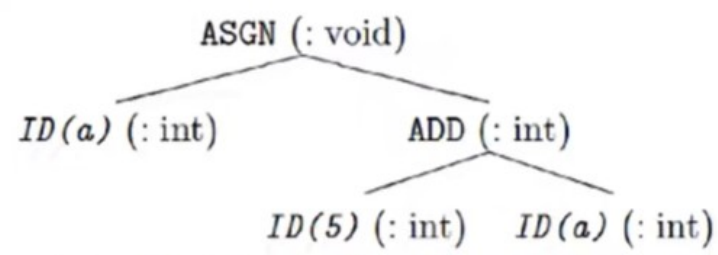
- Type checking is the validation of the set of type rules
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has Boolean type

Type Checking

- More examples:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned
  - For each call statement foo(v_1 ,⋯ , v_n), the type of each actual argument v_i must match the type of the corresponding formal parameter f_i from the declaration of function foo
  - The type of the return value must match the return type from the declaration of the function

Output from Static Semantic Analysis

- Input is AST; output is an annotated tree: identifiers decorated with declarations, other expressions with type information.
- Approximately like the following



- As we shall see, in reality it is usually a bit more involved than this.
  - Normally the annotations are kept in a separate data structure.
  - Things are also complicated because of the issue of *scope*.

Introducing the symbol table

- Semantic checks refer to properties of identifiers in the program — their scope or type
- Need an environment to store the information about identifiers: a *symbol table*
- Each entry in the symbol table contains
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, ...

Name	Kind	Type	Other
`foo`	function	int x int → bool	external
`m`	parameter	int	auto
`n`	parameter	int	const
`tmp`	variable	bool	const

- But this is complicated by the notion of scope.

Scope Information

Scope information characterises the declaration of identifiers and the portions of the program where use of each identifier is allowed
- Example identifiers: variables, functions, objects, labels

Lexical scope is a textual region in the program
- Statement block
- Formal argument list
- Object body
- Function or method body
- Module body
- Whole program (multiple modules)

Scope of an identifier the lexical scope in which it is valid

Scope Information (continued)

- Scope of variables in statement blocks:

```
{ int a;
    ...       \\ in scope of a
    { int b; \\ in scope of a and b
      ...     \\ in scope of a and b
    }         \\ in scope of a and b
    ...       \\ in scope of a
}
```

- *In C:*
  - Scope of file static variables: *current file*
  - Scope of external variables: *whole program*
  - Scope of automatic variables, formal parameters, and function static variables: *the function*

Scope Information (continued)

- Scope of formal arguments of functions/methods:

```
int factorial(int n) {
    ...
}
```

scope of formal parameter `n`?

- Scope of labels:

```
void f() {
    ... goto l; ...
l: a =1;
    ... goto l; ...
}
```

scope of label `l`?

Object fields and methods

- Scope of object fields and methods:

```
class A {
    private int x;
    public void g() { x=1; }
    ...
}
class B extends A {
    ...
    public int h() { g(); }
    ...
}
```

- Scope of field x?
- Scope of method g?

Semantic Rules for Scope

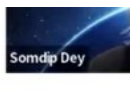
Semantic Rules for Scope

- Main rules regarding scopes:
  - Rule 1 Use an identifier only if defined in enclosing scope
  - Rule 2 Do not declare identifiers of the same kind with identical names more than once in the same scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds
- But might want to avoid this:

```
class X {
   int X;
   void X(int X) {
      X: for(;;)
      break X;
   }
}
```

```
int X(int X) {
  int X;
  goto X;
  { int X;
    X: X = 1; }
}
```

When to perform static checking?



- A more structured approach is to separate out the distinct phases of the compiler.
- We can think about ***performing semantic checks*** *only after the AST has been built*, not while it is changing.
- This approach is more flexible, and less error prone.
- (Arguably for a compiler it is better to concentrate on reliability and correctness than on efficiency, especially given the performance of modern computers.)

Let's explore Static and Dynamic Properties

Static Properties

- Static properties are those properties that can be determined by examining the text of the program without actually "executing" it.
- For example in

  x:=x+z;

  z:=x+2;
- we know that
  - all uses of x (on the RHS) refer to the same variable,
  - the value of the second reference to x is determined by the first assignment statement
  - the final value of z depends on the initial value of x and itself.

  (Henceforth, we can examine the program without executing it)

Dynamic Properties

- Dynamic properties are those properties that can (only) be determined by considering (all possible) executions of the program.
- For example, consider
  - x:=x/z;
    will this trigger an error?
  - x:=y-2;
    if x<2 then z:=10;
    what is the value of z in x:=x/z?
- In general, dynamic properties are not computable

Dynamic v Static

- Sometimes the border between dynamic and static is a bit vague.
  given
  x := 1e-2;
  we might deduce statically, or dynamically that the type of x is float
- In some cases, we can conservatively deduce apparently dynamic properties using static techniques (e.g. using “symbolic execution”).
- “Symbolic execution” is a means of analyzing a program to determine what inputs cause each part of a program to execute

Tasks for a “Semantic” Analyzer

Tasks for a “Semantic” Analyser

- Find the declaration that defines each identifier instance
- Determine the static types of expressions
- Perform re-organizations of the AST that were inconvenient in parser, or required semantic information
- Detect errors and fix to allow further processing

Typical “Semantic” Errors

Multiple declarations a variable should be declared (in the same region) at most once

Undeclared variable a variable should not be used without being declared.

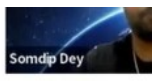
Type mismatch e.g., type of the left-hand side of an assignment should match the type of the right-hand side.

Wrong arguments methods should be called with the right number and types of arguments.

Definite-assignment check (Java): conservative check that simple variables are assigned to before use.

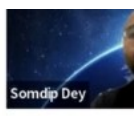
Semantics (part 2), Types and Intermediate languages

Recap – Semantic analysis



- Semantic analysis ensures that the program satisfies a set of additional rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - ▶ Variables must be declared before being used
  - ▶ A variable should not be declared multiple times in the same scope
  - ▶ In an assignment statement, the variable and the assigned expression must have the same type
  - ▶ The condition of an if-statement must have type Boolean
- Some of the categories of semantic rules:
  - ▶ Semantic rules regarding types
  - ▶ Semantic rules regarding scopes

Recap – Static vs Dynamic semantic properties



- Static properties are those properties that can be determined by examining the text of the program without actually "executing" it.
- Dynamic properties are those properties that can (only) be determined by considering (all possible) executions of the program.
- In some cases, we can conservatively deduce apparently dynamic properties using static techniques (e.g. using "symbolic execution").
- "Symbolic execution" is a means of analyzing a program to determine what inputs cause each part of a program to execute

Output from Static Semantic Analysis

- Input is AST; output is an annotated tree: identifiers decorated with declarations, other expressions with type information.
- The annotations are links, or keys, to a symbol table
- The symbol table may include overt information about the scope, or it may be implicit
- A symbol-table look-up can give a different result depending on from where in the program/AST is referenced, to implement the desire for narrow scope declarations to override declarations in the wider scope.
- This may be achieved using some form of scope identifier, or a hierarchy of symbol tables, or stack-based symbol tables, which push on declarations in a given scope, and pop them off when the "current" scope ends.

Output from Static Semantic Analysis

- The details of the implementation depend on how the table is constructed, and used.
- If the annotations are added during tree construction then a stack-based approach may be more natural.
- If constructed afterwards, it might be more convenient to build a tree of symbol tables.
- If it is to be attached to the executable for debugging, then a flat table with explicit scoping contexts might be easier.

Output from Static Semantic Analysis: Cla

- In statically typed languages, symbol entries for classes would contain dictionaries mapping attribute names to types.
- Not all OO languages are statically typed (e.g. consider Python) so the types of the attributes of an object cannot always be typed checked from the text of the program.

Scoping Rules (Names and Referents)

Scoping Rules (Names and Referents)

- Scope of a declaration: section of text or program execution in which declaration applies
- Declarative region: section of text or program execution that bounds scopes of declarations (we'll say "region" for short).
- If scope of a declaration defined entirely according to its position in source text of a program, we say language is statically scoped.
- If scope of a declaration depends on what statements get executed during a particular run of the program, we say language is dynamically scoped.

Scoping Rules (Name Reuse and Nesting)

- In most languages, can declare the same name multiple times, if its declarations
  - ▶ occur in different declarative regions, or
  - ▶ involve different kinds of names.
    Examples from Java?, C++?
- Nesting:
  - ▶ Most statically scoped languages (including C, C++, Java) use "Algol scope rule": Where multiple declarations might apply, choose the one defined in the innermost (most deeply nested) declarative region.
  - ▶ Often expressed as "inner declarations hide outer ones."
  - ▶ Variations on this: Java disallows attempts to hide local variables and parameters.

Scoping Rules (Declarative Regions)

- Languages differ in their definitions of declarative regions.
- In Java, variable declaration's effect stops at the closing '}', that is, each function body is a declarative region.
- What about others?
- In Python, function header and body make up a declarative region, as does a lambda expression. But nothing smaller.
- There is Just one x in this program:

```
def f(x):
        x= 3
        L = [x for x in xrange(0,10)]
```

Scoping Rules (“use” before “declaration”)

- Languages have taken various decisions on where scopes start.
- In Java, C++, scope of a member (field or method) includes the entire class (textual uses may precede declaration).
- But scope of a local variable starts at its declaration.
- As for non-member and class declarations in C++ must write:

```
extern int f(int);
class C;
int x = f(3);
void g(C* x) {
...
}
// Forward declarations
// Would be illegal w/o forward decls. int f (int x) { ... }
class C { ... }
```

Scoping Rules (Overloading)

- In Java or C++ (not Python or C), can use the same name for more than one method, as long as the number or types of parameters are unique.

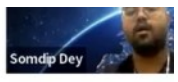
```
int add(int a, int b);
float add(float a, float b);
```

- The declaration applies to the signature —name & argument types— not just name.
- But return type not part of signature, so this won’t work:

```
int add (int a, int b);
    float add (int a, int b)
```

- In Ada (a statically typed object-oriented high-level programming language), it will, because the return type is part of signature.
- (There is also the issue of name spaces.)

Dynamic Scoping

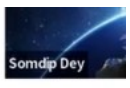


- Original Lisp, APL, Snobol use dynamic scoping, rather than static: Use of a variable refers to most recently executed, and still active, declaration of that variable.
- Makes static determination of declaration generally impossible.
- Example:

```
void main() { f1(); f2(); }
void f1() { int x = 10; g(); }
void f2() { String x = "hello"; f3();g(); }
```

- With static scoping, illegal.
- With dynamic scoping, prints “10” and “hello”

Implicit v Explicit Declaration



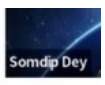
- Java, C++ require explicit declarations of things.
- C is lenient: if you write foo(3) with no declaration of foo in
- scope, C will supply one.
- Python implicitly declares variables you assign to in a function to be local variables.
- Fortran implicitly declares any variables you use, and gives them a type depending on their first letter.
- But in all these cases, there is a declaration as far as the compiler is concerned.

Why do we need Symbol Table: Decorating the AST

- Idea is to recursively navigate the AST,
  - ►in effect executing the program in simplified fashion,
  - ►extracting information that isn't data dependent.

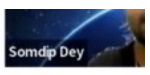
Types & Type Checking

Why do we need Type Checking Phase



- Type checking phase determines the type of each expression in the program, (i.e. each node in the AST that corresponds to an expression)
- The type rules of a language define each expression's type and the types required of all expressions and subexpressions.

Uses of Types



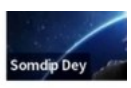
Detect errors:

- ►Memory errors, such as attempting to use an integer as a pointer.
- ►Violations of abstraction boundaries, such as using a private field from outside a class.

Help compilation:

- ►When Python sees x+y, its type systems tells it almost nothing about types of x and y, so code must be general.
- ►In C, C++, Java, code sequences for x+y are smaller and faster, because representations are known.

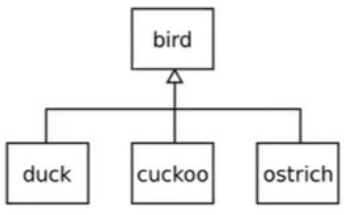
Types and Type Systems



- A type is a set of values together with a set of operations on those values.
- E.g., fields and methods of a Java class are meant to correspond to values and operations.
- A language's type system specifies which operations are valid for which types.
- Goal of type checking is to ensure that operations are used with the correct types, enforcing intended interpretation of values.
- Notion of "correctness" often depends on what programmer has in mind, rather than what the representation would allow.
- Most operations are legal only for values of some types
  - ►Doesn't make sense to add a function pointer and an integer in C
  - ►It does make sense to add two integers
  - ► But both have the same assembly language implementation denoted by: movl y, %eax; addl x, %eax

Subtyping

- In **programming language** theory, **subtyping** is a form of type polymorphism in which a **subtype** is a datatype that is related to another datatype (the supertype) by some notion of substitutability.
- This means that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of that subtype.
- A simple practical example of subtypes is shown in the diagram using classes.
- The type "bird" has three subtypes "duck", "cuckoo" and "ostrich". Conceptually, each of these is a variety of the basic type "bird" that inherits many "bird" characteristics but has some specific differences.



Typing Options

Statically typed almost all type checking occurs at compilation time (C, Java). Static type system is typically rich. (Can be further subdivided into *weak* and *strong* typing.)

Dynamically typed almost all type checking occurs at program execution (Scheme, Python, Javascript, Ruby). Static type system can be trivial.

Untyped no type checking. What we might think of as type errors show up either as weird results or as various runtime exceptions.

"Type Wars"

- Dynamic typing proponents say:
  - Static type systems are restrictive; can require more work to do reasonable things.
  - Rapid prototyping easier in a dynamic type system.
  - Use *duck typing*: define types of things by what operations they respond to ("if it walks like a duck and quacks like a duck, it's a duck").
- Static typing proponents say:
  - Static checking catches many programming errors at compile time.
  - Avoids overhead of runtime type checks.
  - Use various devices to recover the flexibility lost by "going static:" *subtyping*, *coercions*, and type *parameterisation*.
  - Of course, each such wrinkle introduces its own complications.
  - Note that *weak* static typing has some of the features of dynamic typing, whilst being safer.

Using Subtypes

- In languages such as Java, it can define types (classes) either to
  - ▶ Implement a type, or
  - ▶ Define the operations on a family of types without (completely) implementing them.
- Hence, this relaxes static typing a bit: we may know that something is a Y without knowing precisely which subtype it has.

Implicit Coercions

- In Java, we can write
  int x = 'c';
  float y = x;
- But relationship between char and int, or int and float are not usually called subtyping, but rather conversion (or coercion).
- Such implicit coercions avoid cumbersome casting operations.
- Might cause a change of value or representation,
- But usually, such coercions allowed implicitly only if type coerced to contains all the values of the that coerced from (a widening coercion).
- Inverses of widening coercions, which typically lose information (e.g., int → char), are known as narrowing coercions. and typically required to be explicit.
- int → float a traditional exception (implicit, but can lose information and is neither a strict widening nor a strict narrowing.)

Let's look at Coercion examples

Object x = ...;
String y = ...;
int a = ...; short b = 42;
x = y; a = b; // OK
y = x; b = a; // widening ERRORS
x = (Object) y; // OK
a = (int) b; // OK
y = (String) x; // OK but may cause exception b = (short) a; // OK but may lose information

- Possibility of implicit coercion complicates type-matching rules (see C++).

Type Inference

- Types of expressions and parameters need not be explicit to have static typing. With the right rules, might *infer* their types.
- The appropriate formalism for type checking is logical rules of inference having the form

  If Hypothesis is true, then Conclusion is true

- For type checking, this might become:

  If E_1 and E_2 have certain types, then E_3 has a certain type.

- Given proper notation, easy to read (with practice), so easy to check that the rules are accurate.
- Can even be mechanically translated into programs.

Other Issues

- There are also control flow errors:
  - ►Must verify that a break or continue statement is always enclosed by a while (or for) statement
  - ► Java: must verify that a break X statement is enclosed by a for loop with label X
  - ►Can easily check control-flow errors by recursively traversing the AST

Use of natural deduction to deduce type

- We write ! ⊢ # (A proves B) indicating that B can be derived from A using some 'correct rules of reasoning'
- This means we can find a derivation from A to B (through a sequence of logical steps) than the statement ! ⊢ B is true
- This is based on natural deduction rules

What Is Natural deduction rules

- There two kinds of rules:

1. The one that tells us how to reason sentential form with connectives. E.g.
   - given ! ∨ # is true means either A or B or both are true. This is rule is called an elimination rule.
2. How to deduce a compound sentence with a given connectives – that is how to prove a conclusion. E.g. to prove ! ∨ # we must prove at least A or B is true, this rule is referred as induction rule.

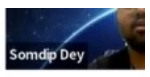
When we write ! ⊢ B, it is called a sequent.

It means a proof exits, from set of data A, to reach at B.

- It is often presented in the following format

$\frac{\%}{\&}$ the actual proof will be a sequence of sub-proofs

Natural deduction rules (Continued)



- The set A of sentences is called antecedents and
- B is called conclusion, we write:

$$\frac{!\ \text{"\#\$\%\&!'\ \$\#(\#)\#'\ \$"}}{!\ (\%'(*+",\%'\quad -+*\#\ +"\#)}$$

- Antecedent basically means "something that has gone before" a prior.
- Sometime we might write:

$$\frac{}{!\ (\%'(*+",\%'\quad -+*\#\ +"\#)}$$

where the antecedent set is empty or simply blank, this means the conclusion arrived without any antecedent, these are usually the axioms of the environment or the system.

An environment

- We often express our logical steps (i.e. the deduction rules) within a context of an environment.
- The gamma environment is the context in which the variables are declared, which basically means that within the environment (i.e. within the program) these rules governs, than a series of rules will be presented that are applicable in the context at hand.

An example of environment

- Environment can be denoted by Γ

AND

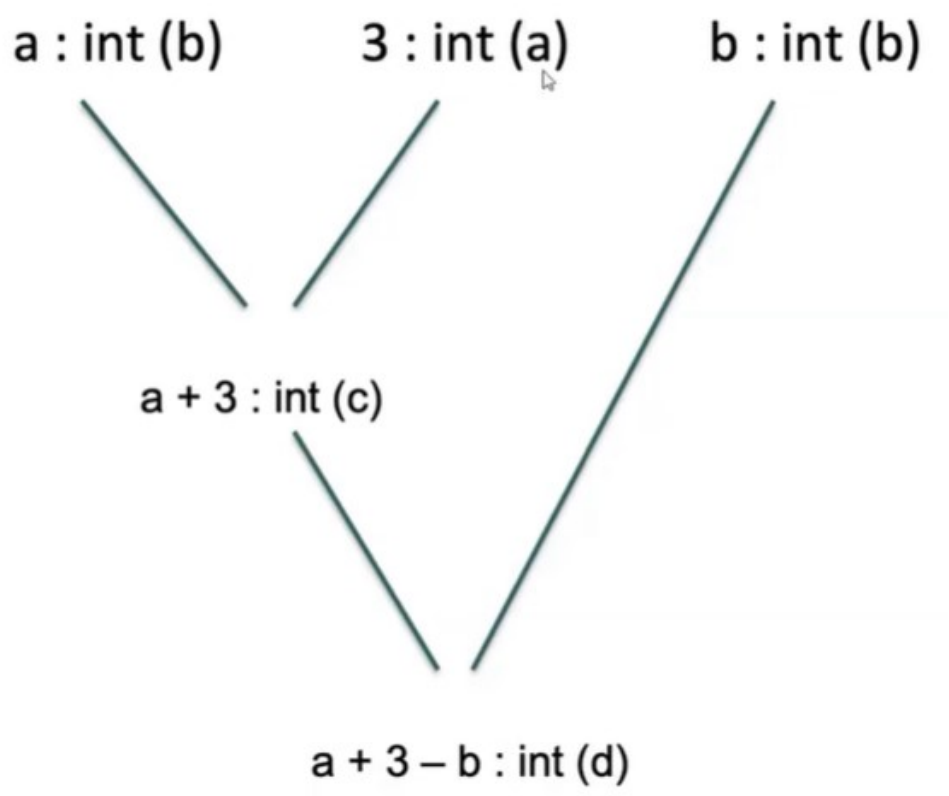
- We might be given:

(a) $\frac{}{\Gamma \vdash <integer>:int}$ (b) $\frac{}{\Gamma, int:v \vdash v:int}$

(c) $\frac{\Gamma \vdash e1:int \quad \Gamma \vdash e2:int}{\Gamma \vdash (e1+ e2):int}$ (d) $\frac{\Gamma \vdash e1:int \quad \Gamma \vdash e2:int}{\Gamma \vdash (e1- e2):int}$

- Then asked:
- Deduce the type of the expression "a + 3 – b"

The Answer Is



Antlr: Using Listener

Antlr: Using Listener

- Previously we used visitor pattern to walk the parse-tree
- Antlr also provide Listener – by default
- Next we are going to look at a small program that uses a listener to calculate/evaluate a simple arithmetic expression using stack.
- We need to create a class that extends the language BaseListener
- BaseListener provides two methods for each component of expr, e.g. enterStart and exitStart where enter when a node is arrived at while exit when the parsing is done

Expression grammar

```
grammar Expr;

start : expr ;
expr    :
          expr op=('*'|'/') expr     # Mult
        |  expr op=('+'|'-') expr     # Add
        |  INT                        # Int
        |  '(' expr ')'               # Paren
        ;

INT    : [0-9]+ ;
WS     : [ \t\r\n]+ -> skip;
```

Antlr Listener -1 Implementation

```
EvalExprListener extends ExprBaseListener

public void exitInt(ExprParser.IntContext context) {
        int i = Integer.valueOf(context.INT().getText());
        stack.push(i);
 }

public void exitStart(ExprParser.StartContext context)  {
        int result = stack.pop();
        System.out.println("result " + result);
}
```

Listener – 2 Implementation

```
public void exitMult(ExprParser.MultContext context)  {
        int r = stack.pop();
        int l = stack.pop();
        String operator = context.op.getText();
        int result;
        if (operator.equals("*"))
           result = l * r;
        else
           result = l / r;
        stack.push(result);
   }
```

Listener – 3 Implementation

```
@Override
    public void exitAdd(ExprParser.AddContext context)  {
       int r = stack.pop();
       int l = stack.pop();
       String operator = context.op.getText();
       int result;
       if (operator.equals("+"))
          result = l + r;
       else
          result = l - r;
       stack.push(result);
    }
```

Intermediate Languages

Intermediate Languages and Machine Languages

- From the parse trees (or AST), we could produce machine language directly.
- However, it is often convenient to first generate some kind of intermediate language (IL): a “high-level machine language” for a “virtual machine.”
- Sometimes called an intermediate representation.

Advantages of Intermediate Languages

- Separates problem of extracting the operational meaning (the dynamic semantics) of a program from the problem of producing good machine code from it, because it ...
- Gives a clean target for code generation from the AST.
- By choosing IL judiciously, we can make the conversion of IL → machine language easier than the direct conversion of AST → machine language. Helpful when we want to target several different architectures (e.g., gcc).
- Likewise, if we can use the same IL for multiple languages, we can re-use the IL → machine language implementation (e.g., gcc, CIL from Microsoft's Common Language Infrastructure).

ASTs — High-Level IRs

- ASTs can be thought of as a high-level, language independent Intermediate Representation.
- The tree node structure is a form of language, as expression nodes and statement nodes common to many languages.

Expression Nodes

- Integers and program variables
- Binary operations: e1 OP e2
  - ▶ Arithmetic operations
  - ▶ Logic operations
  - ▶ Comparisons
- Unary operations: OP e
- Array accesses: e1[e2]

Statement Nodes

- Block statements (statement sequences): `(s1, ..., sN)`
- Variable assignments: `v = e`
- Array assignments: `e1[e2] = e3`
- If-then-else statements: `if c then s1 else s2`
- If-then statements: `if c then s`
- While loops: `while (c) s`
- Function call statements: `f(e1, ..., eN)`
- Return statements: `return` or `return e`

Other Constructs

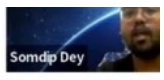
- For loop statements: for(v = e1 to e2) s
- Break and continue statements
- Switch statements: switch(e) v1: s1, ..., vN: sN

Low-level Intermediate Representations

Low-level Intermediate Representations

- Low-level representation is essentially an instruction set for an abstract machine
- Alternatives for low-level IR:
  - ▶ Three-address code or quadruples (Dragon Book): a = b OP c
  - ▶ Tree representation (Tiger Book “Modern Compiler Implementation in Java/ML/C” Appel)
  - ▶ Stack machine (like Java bytecode)

Stack Machines as Virtual Machines



- A **stack machine** is a mode of computation where executive control is maintained wholly through append (push), readoff and truncation (pop), of a first-in-last-out (FILO, also last-in-first-out or LIFO) memory buffer, known as a stack, requiring very few processor registers.
- It is a simple evaluation model: instead of registers, a stack of values for intermediate results.
- Examples: The Java Virtual Machine, the Postscript interpreter,
- . . . and Forth.
- Each operation (1) pops its operands from the top of the stack, (2) computes the required operation on them, and (3) pushes the result on the stack.

Ex: A program to compute 7 + 5

```
push 7 # Push constant 7 on stack
push 5 # Push constant 5 on stack
add # Pop 5 and 7 from stack, add, and push result.
```

Advantages of Stack-based ILs

- Uniform compilation scheme: Each operation takes operands from the same place and puts results in the same place.
- This leads to a uniform compilation scheme (and hence a simpler compiler).
- The location of operands is implicit (always on the top of the stack).
- No need to specify the location of the result.
- Add instruction is just add, rather than add r1 r2.
- Fewer explicit operands in instructions means smaller encoding of instructions and more compact programs. (Hence its use in Java bytecode, and the CLR.)
- Meshes nicely with subroutine calling conventions that push arguments on stack.

Stack Machine with Accumulator

- The *add* instruction performs three memory operations: Two reads and one write of the stack.
- The top of the stack is frequently accessed
- Idea: keep most recently computed value in a register (called the *accumulator*) since register accesses are faster.
- For an operation $op(e_1, \ldots, e_n)$:
  - compute each of $e_1, \ldots, e_{n-1}$ into *acc* and then push on the stack;
  - compute e_n into the accumulator;
  - perform op computation, with result in *acc*.
  - Pop $e_1, \ldots, e_{n-1}$ off stack.
- Expression evaluation preserves the stack.

Example of Stack Machine with Accumula

- The *add* instruction is now

  $acc \leftarrow acc + top\_of\_stack$
  pop one item off the stack

  and uses just one memory operation (popping just means adding constant to stack-pointer register).
- After computing an expression the stack is as it was before computing the operands.

Example: Full computation of 7+5

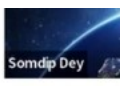
$acc \leftarrow 7$
$push\ acc$
$acc \leftarrow 5$
$acc \leftarrow acc + top\_of\_stack$
$pop\ stack$

A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	⟨init⟩
push acc	3	3 ⟨init⟩
acc ← 7	7	3 ⟨init⟩
push acc	7	7 3 ⟨init⟩
acc ← 5	5	7 3 ⟨init⟩
acc ← acc + top_of_stack	12	7 3 ⟨init⟩
pop	12	3 ⟨init⟩
acc ← acc + top_of_stack	15	3 ⟨init⟩
pop	15	⟨init⟩

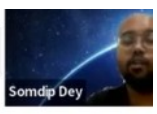
Reverse polish (aka Postfix)

Reverse polish



- The operator follows the operands
- For example 2 + 3 in reverse polish becomes 2 3 +
- 3 + 2 – 1 in reverse polish 3 2 + 1 –
- First 2 is add to 3, then 1 is subtracted from it.
- In reverse polish we don't need parenthesis
- For example 3 + 2 * 1 which mean 3 + (2 * 1) in reverse polish is 3 2 1 * +
- While (3 + 2) * 1 in reverse polish 3 2 + 1 *
- Consequently, as you can observe, in reverse polish there is no need for parenthesis

How to change Pprint to print Reverse-Pol (Code)




```
@Override
public String visitMultiplicative(EParser.MultiplicativeContext
ctx) {
    String left = visit(ctx.left);
    String right = visit(ctx.right);
    String operator = ctx.op.getText();
    if (operator.equals("*"))
        return left + " " + right + " * ";
    else
        return left + " " + right + " / ";
}
```

Reverse-Polish Code Continued

```
@Override
public String visitAdditive(EParser.AdditiveContext ctx)
{
    String left = visit(ctx.left);
    String right = visit(ctx.right);
    String operator = ctx.op.getText();
    if (operator.equals("+"))
        return left + " " + right + " + ";
    else
        return left + " " + right + " - ";
}
```

Reverse-Polish Code Continued

```
@Override public String
visitParentheses(EParser.ParenthesesContext ctx) {

    return this.visit(ctx.e()) ;
}
```

- Just remove the parentheses.