Reminder: How Does a Computer Run Your C Program?

- You edit `myprogram.c`
- You **compile:** `cc -o myprogram myprogram.c`
  - **Preprocessor** generates **preprocessed source** (`myprogram.i`)
  - **Compiler proper** generates **assembly program** (`myprogram.s`)
  - **Assembler** generates **object code** (`myprogram.o`)
  - **Linker** generates **executable** (`myprogram`)
- You "**run**" it: `./myprogram`
  - **Operating system** generates a new process
  - **Dynamic linker** resolves references to shared libraries
  - **Loader** generates **executable in-memory image**
  - **CPU** runs machine code

Programming Language Implementation

Translation

Source language programs are translated into *target language* programs:

- **Assembler:** symbolic representation of machine code $\rightarrow$ machine code
- **Compiler:** high(er)-level language $\rightarrow$ low(er)-level language
- **Loader / link editor** translates address references in object code indicated by address tables to actual addresses
- **Macroprocessor / preprocessor** performs macro expansion and code fragment selection by applying rewriting rules

Software simulation — Virtual machines

Create a (low-level) program that acts as a "computer whose machine language is the high-level language".

This **interpreter** also acts as a **virtual machine** implementation.

Most "interpreters" first perform compilation into some internal representation (sometimes exported as **bytecode**).

Stages in Translating a Program

Lexical analysis (Scanner): Breaking a program into primitive components, called **tokens** (identifiers, numbers, keywords, …)

Syntactic analysis (Parsing): Creating a syntax tree of the program.

Symbol table: Storing information about declared objects (identifiers, procedure names, …)

Semantic analysis: Understanding the relationship among the tokens in the program.

Optimization: Rewriting the syntax tree to create a more efficient program.

Code generation: Converting the parsed program into an executable form.

Each stage is based on a specification of the relevant language aspect!

Describing Programming Languages

Syntax — *Shape* of PL constructs

- What are the **tokens** of the language? — **Lexical** syntax, "word level"
- How are programs built from tokens? — Mostly use **Context-Free Grammars** (CFG) or **Backus-Naur-Form** (BNF) to describe **syntax** at the "sentence level"

Semantics — *Meaning* of PL constructs

- Three major approaches to PL semantics:
  - **Axiomatic semantics:** $\{p\}$ `Prog` $\{q\}$
  - **Denotational semantics:** `Prog` *denotes* a *mathematical* function $[\![$`Prog`$]\!]$
  - **Operational semantics:** state transition sequence of an abstract machine
- **"Static semantics":** aspects of program structure that are checked at compile time, but cannot be captured by CFGs ($\rightarrow$ context-sensitive syntax):
  - Scopes of names
  - Typing

Formal Languages, Grammars, Automata

A **formal language** *over an alphabet A* is a subset of A^*.

Formal languages can be *generated* by **grammars**, *recognized* by **automata**.

Phase	*Input Alphabet*	*Output*	*Grammar Type*	*Recognising Automata*	*Generators*
Lexing	Characters	Token **Sequence**	Type 3: Regular	Finite Automata	`lex, flex ocamllex alex`
Parsing	Tokens	Syntax **Tree**	Type 2: Context-Free	Pushdown Automata	`yacc, bison ANTLR, JavaCC ocamlyacc, happy`

Two levels of formal languages:

- **token languages** over character-level alphabet
- **program language** over token alphabet

Token Example: Identifiers in Java

Java 2 Language Spec. 3.8:

```
IdentifierChars:
        JavaLetter
        IdentifierChars JavaLetterOrDigit
```

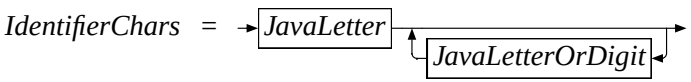
Conventional BNF:

```
IdentifierChars ::=  JavaLetter
                  |  IdentifierChars JavaLetterOrDigit
```

Conventional CFG:

```
IdentifierChars  →  JavaLetter
IdentifierChars  →  IdentifierChars JavaLetterOrDigit
```

"Railroad diagram":



Regular Expression:

$$\mathit{IdentifierChars} = \mathit{JavaLetter} \cdot \mathit{JavaLetterOrDigit}^*$$

BNF in the Textbook

```
Integer  →  Digit | Integer Digit
Digit    → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This is an abbreviation for the following set of CFG rules:

```
Integer  →  Digit
Integer  →  Integer Digit
Digit    → 0
Digit    → 1
           ⋮
Digit    → 9
```

Definition: A **context-free grammar (CFG)** is a tuple (Σ, N, S, ρ) where

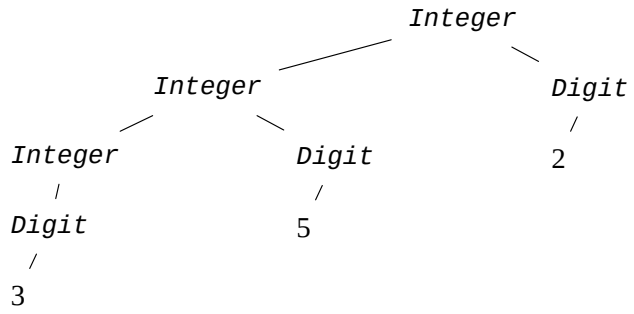
- Σ is a set of **terminal symbols**
- N is a set of **nonterminal symbols**
- $S \in N$ is the **start** symbol
- $\rho \subseteq (N \times (N \cup \Sigma)^*)$ is a set of rules;

  a rule (A, ω) is usually written "$A \to \omega$"

Derivations and Parse Trees

$$\mathit{Integer} \to \mathit{Integer\ Digit} \to \mathit{Integer\ Digit\ Digit}$$
$$\to \mathit{Digit\ Digit\ Digit} \to 3\,\mathit{Digit\ Digit} \to 35\,\mathit{Digit} \to 352$$

$$\mathit{Integer} \to \mathit{Integer\ Digit} \to \mathit{Integer}\,2$$
$$\to \mathit{Integer\ Digit}\,2 \to \mathit{Integer}\,52 \to \mathit{Digit}\,52 \to 352$$



Regular Grammars

If all productions are of shape $N_1 \rightarrow t$ or $N_1 \rightarrow N_2 t$, then the grammar is called **regular**.

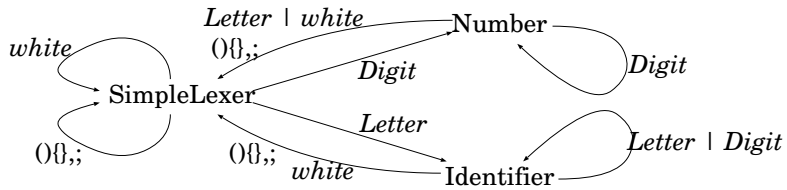
InputElement	→	*WhiteSpace* | *Comment* | *Token*
WhiteSpace	→	' ' | \t | \r | \n | \f | \r\n
Token	→	*Identifier* | *Keyword* | *Literal* | *Separator* | *Operator*
Identifier	→	*Letter* | *Identifier Letter* | *Identifier Digit*
Letter	→	`a` | `b` | … | `z` | `A` | `B` | … | `Z`
Digit	→	0 | 1 | … | 9
Keyword	→	`boolean` | `else` | `if` | `int` | `main` | `void` | `while`
Separator	→	`(` | `)` | `{` | `}` | `;` | `,`
		⋮

Lexical Analysis

- The languages ***generated by regular grammars*** are precisely the languages ***accepted by finite-state automata***.
- These languages are called **regular languages**
- Lexical syntax is defined as a set of **token classes**
- Each token class corresponds to a regular language (typically all disjoint)
- Lexical analysis: find out which tokenclass contains a prefix of the character stream

Regular Grammars — Simplified Example

InputElement	→	*WhiteSpace* | *Token*
WhiteSpace	→	' ' | \t | \r | \n | \f | \r\n
Token	→	*Identifier* | *Number* | *Separator*
Identifier	→	*Letter* | *Identifier Letter* | *Identifier Digit*
Number	→	*Digit* | *Number Digit*
Letter	→	`a` | `b` | … | `z` | `A` | `B` | … | `Z`
Digit	→	0 | 1 | … | 9
Separator	→	`(` | `)` | `{` | `}` | `;` | `,`



Lexical Analysis — Java Example

Making evaluation-on-demand explicit:
- calling *nextToken*() demands next token
- *nextToken*() calls *readChar*() to demand characters

```
class Token {
   public String type;     // token type: Identifier, Number, Separator, or Other
   public String value;    // token value
}

public class TokenStream {
   private boolean isEof = false;
   private char nextChar = ' ';     // next character in input stream
   private BufferedReader input;

   private char readChar () {    ...   // details omitted
      try { i = input.read(); } ...   // details omitted
      if (i == –1) { isEof=true; return (char)0; }
      return (char)i;
   }
```

Lexical Analysis — Java Example (ctd.)

```
public Token nextToken() { // Return next token type and value
  Token t = new Token();
  t.type = "Other"; t.value = "";

  // first check for whitespace and bypass it
  while (isWhiteSpace(nextChar)) { nextChar = readChar(); }

  // Then check for a Separator
  if (isSep(nextChar)) {
     t.type = "Separator";
     t.value = t.value + nextChar;
     nextChar = readChar();
     return t;
  }

  // Then check for an Identifier
  if (isLetter(nextChar)) { // get an Identifier
     t.type = "Identifier";
     while ((isLetter(nextChar) || isDigit(nextChar))) {
        t.value = t.value + nextChar;
        nextChar = readChar();
     }
```


Regular Expressions

Definition: A **regular expression** over an alphabet Σ is

- ε, standing for the empty string
- an element of Σ
- alternative $M \mid N$ of two regular expressions M and N
- concatenation MN of two regular expressions M and N
- iteration M^* of a regular expressions M

Each regular expression denotes a **regular language**:

- $\varepsilon = \{\langle\rangle\}$
- If $a \in \Sigma$, then $a = \{\langle a \rangle\}$
- $M \mid N = M \cup N$ — union of languages
- $MN = M \cdot N$ — concatenation of languages
- $M^* = \bigcup_{i \in \mathbb{N}} M^i$

Extended Regular Expressions

- $M^+ \equiv MM^* = \bigcup_{i \in \mathbb{N} \setminus \{0\}} M^i$
- $M? \equiv M \mid \varepsilon$
- $[a\text{-}z] \equiv a \mid b \mid c \mid \cdots \mid y \mid z$ — requires a linear ordering on Σ
- $[a\text{-}zA\text{-}Z] \equiv [a\text{-}z] \mid [A\text{-}Z]$
- $. = \Sigma$
- $[\hat{}a\text{-}z] = \Sigma \setminus [a\text{-}z]$

– Read the UNIX manual pages for **`grep`** and **`egrep`**; compare the regular expressions there with those here and with those in the textbook.
– Learn what **`awk`** and **`sed`** are used for (UNIX texts, manual pages), and what the basic structure of **`awk`** and **`sed`** scripts is.
– Have you ever encountered any problems that you now would solve using **`grep`** **`awk`**, and **`sed`**?

Regular Expression Examples

- $Nat = [0\text{-}9]^+$
- $Integer = \text{-}?[0\text{-}9]^+$
- $Identifier = [a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$
- $LineComment = //[\hat{}\backslash r \backslash n \backslash t]^*[\backslash r \backslash n \backslash t]$

Lexer Generators convert regular expression token definitions into efficient implementations of finite-state automata

– **`lex`**, **`flex`**, **`Jlex`**, **`Alex`**, **`ocamllex`**, …

Regular Expressions vs. Context-Free Grammars

Definition: A **regular expression** over an alphabet Σ is

- ε, standing for the empty string
- an element of Σ
- alternative $M \mid N$ of two regular expressions M and N
- concatenation MN of two regular expressions M and N
- iteration M^* of a regular expressions M

Definition: A **context-free grammar (CFG)** is a tuple (Σ, N, S, ρ) where

- Σ is a set of **terminal symbols**
- N is a set of **nonterminal symbols**
- $S \in N$ is the **start** symbol
- $\rho \subseteq (N \times (N \cup \Sigma)^*)$ is a set of rules;
  a rule (A, ω) is usually written "$A \to \omega$"

Regular Languages vs. Context-Free Languages

A language is **regular** iff there is a regular expression denoting it

- ***Fact:*** A language is regular iff there is a DFA accepting it
- ***Fact:*** A language is regular iff there is a NFA accepting it

A language is **context-free** iff there is a context-free grammar generating it

- ***Fact:*** A language is **context-free** iff there is a pushdown-automaton (≈ NFA with stack) accepting it
- ***Fact:*** **All regular lanugages are context-free**
- ***Fact:*** **Many context-free languages are not regular**

  Examples:
  - $\{a^n b^n\} = \bigcup_{n:\mathbb{N}} a^n b^n$
  - Expression languages with matching parentheses nested to arbitrary depth
  - Palindromes

An Expression Grammar

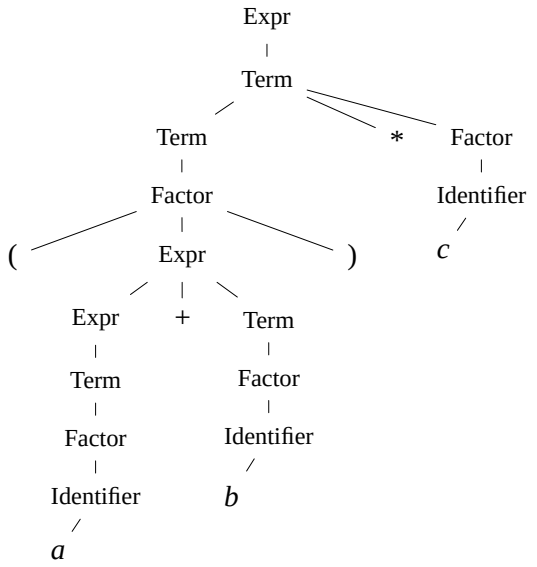
Assignment	→	*Identifier* = *Expression*
Expression	→	*Term* | *Expression* + *Term* | *Expression* − *Term*
Term	→	*Factor* | *Term* ∗ *Factor* | *Term* / *Factor*
Factor	→	*Identifier* | *Literal* | (*Expression*)

---

48 / y + x 48 / (y + x) 48 / (y / x) 48 / y / x (48 / y) / x

Parsing "$(a + b) * c$"

Expr	→	*Term*
	|	*Expr* + *Term*
	|	*Expr* − *Term*
Term	→	*Factor*
	|	*Term* ∗ *Factor*
	|	*Term* / *Factor*
Factor	→	*Ident*
	|	*Literal*
	|	(*Expr*)



The (simplified) Jay Grammar in BNF (Textbook: B.2.2)

Program	→	`void main ( ) {` *Declarations Statements* `}`
Declarations	→	ε | *Declarations Declaration*
Declaration	→	*Type Identifiers* `;`
Type	→	`int` | `boolean`
Statements	→	ε | *StatementsStatement*
Statement	→	`;` | *Block* | *Assignment* | *IfStmt* | *WhileStmt*
Block	→	`{` *Statements* `}`

---

void *main* ( ) { int *x* ; *x* = 1; }

Ambiguity

Exp → *Integer* | *Exp* + *Exp* | *Exp* − *Exp* | *Exp* ∗ *Exp* | *Exp*/*Exp*

48 / 6 / 2

48 / 6 / 2

Programming language grammars should not be ambiguous!

Dangling `else`

IfStatement	→	`if (` *Expression* `)` *Statement*
	|	`if (` *Expression* `)` *Statement* `else` *Statement*

```
if( x<0 ) if( y<0 ) y=y-1; else y=0;
```

```
if( x<0 ) if( y<0 ) y=y-1; else y=0;
```

Solutions:

- non-CFG rules — C, C++
- extra non-terminal *StatementNoShortIf* — Java
- `end if`, `endif`, `fi` — Ada
- no “short if” — Haskell

EBNF — Extended BNF

Right-hand sides of productions are:

- **strings** of terminal and non-terminal symbols in **conventional CFGs**
- **alternatives** of such strings in **BNF**
- **regular** expressions over terminal and non-terminal symbols in **EBNF**

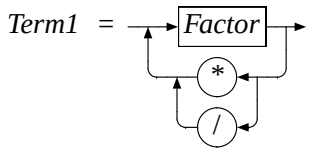
All three variants have the **same expressivity**:

- BNF can be directly translated into CFGs by expanding alternatives into sets of rules for the same non-terminal
- Transformation of EBNF into BNF may require additional non-terminals
- Certain EBNF grammars directly induce **recursive descent parsers**

The (simplified) Jay Grammar in EBNF (Textbook: B.2.1)

Program	→	`void main ( )` '`{`' *Declarations Statements* '`}`'
Declarations	→	{*Declaration*}*
Declaration	→	*Type Identifiers* `;`
Type	→	`int` | `boolean`
Statements	→	{*Statement*}*
Statement	→	`;` | *Block* | *Assignment* | *IfStmt* | *WhileStmt*
Block	→	'`{`' *Statements* '`}`'
Assignment	→	*Identifier* = *Expression* `;`
Addition	→	*Term*1 {[+ | −]*Term*1}*
*Term*1	→	*Factor* {[∗ | /]*Factor*}*
Factor	→	*Identifier* | *Literal* | `(` *Expression* `)`

"**Railroad diagram**":



Java Expression Grammar (Fragment)

```
Primary:
    PrimaryNoNewArray
    ArrayCreationExpression

PrimaryNoNewArray:
    Literal
    Type . class
    void . class
    this
    ClassName.this
    ( Expression )
    ClassInstanceCreationExpression
    FieldAccess
    MethodInvocation
    ArrayAccess

ClassInstanceCreationExpression:
    new ClassOrInterfaceType ( ArgumentList_opt )
                  ClassBody_opt
    Primary.new Identifier ( ArgumentList_opt )
                  ClassBody_opt

ArgumentList:
    Expression
    ArgumentList , Expression

FieldAccess:
    Primary . Identifier
    super . Identifier
    ClassName .super . Identifier

MethodInvocation:
    MethodName ( ArgumentList_opt )
    Primary . Identifier ( ArgumentList_opt )
    super . Identifier ( ArgumentList_opt )
    ClassName . super . Identifier ( ArgumentList_opt )
```

```
PostfixExpression:
    Primary
    ExpressionName
    PostIncrementExpression
    PostDecrementExpression

UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression
    + UnaryExpression
    - UnaryExpression
    UnaryExpressionNotPlusMinus

PreIncrementExpression:
    ++ UnaryExpression

PreDecrementExpression:
    -- UnaryExpression

UnaryExpressionNotPlusMinus:
    PostfixExpression
    ~ UnaryExpression
    ! UnaryExpression
    CastExpression

CastExpression:
    ( PrimitiveType ) UnaryExpression
    ( ReferenceType ) UnaryExpressionNotPlusMinus

MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
```


Haskell Expression Grammar

exp	→	exp^0 `::` [*context* `=>`] *type*
	|	exp^0
exp^i	→	exp^{i+1} [$qop^{(n,i)}$ exp^{i+1}]
	|	$lexp^i$
	|	$rexp^i$
$lexp^i$	→	($lexp^i$ | exp^{i+1}) $qop^{(l,i)}$ exp^{i+1}
$lexp^6$	→	`-` exp^7
$rexp^i$	→	exp^{i+1} $qop^{(r,i)}$ ($rexp^i$ | exp^{i+1})
exp^{10}	→	`\`$apat_1 \ldots apat_n$`->` *exp*
	|	`let` *decls* `in` *exp*
	|	`if` *exp* `then` *exp* `else` *exp*
	|	`case` *exp* `of` {*alts*}
	|	`do` {*stmts*}
	|	*fexp*

fexp	→	[*fexp*] *aexp*
aexp	→	*qvar*
	|	*gcon*
	|	*literal*
	|	`(` *exp* `)`
	|	`(` exp_1`,` … `,` exp_k `)`
	|	`[` exp_1`,` … `,` exp_k `]`
	|	`[` exp_1`[,` exp_2`]` `..` `[` exp_3`]` `]`
	|	`[` *exp* `|` $qual_1, \ldots, qual_n$ `]`
	|	`(` exp^{i+1} $qop^{(a,i)}$ `)`
	|	`(` $lexp^i$ $qop^{(l,i)}$ `)`
	|	`(` $qop^{(a,i)}_{\langle\cdot\rangle}$ exp^{i+1} `)`
	|	`(` $qop^{(r,i)}_{\langle\cdot\rangle}$ $rexp^i$ `)`
	|	*qcon* `{` $fbind_1, \ldots, fbind_n$ `}`
	|	$aexp_{\langle qcon\rangle}$ `{` $fbind1, \ldots, fbindn$ `}`

Commented Haskell Expression Grammar

exp	→	exp^0 `::` [*context* `=>`] *type*	(expression type signature)
	|	exp^0	
exp^i	→	exp^{i+1} [$qop^{(n,i)}$ exp^{i+1}]	
	|	$lexp^i$	
	|	$rexp^i$	
$lexp^i$	→	($lexp^i$ | exp^{i+1}) $qop^{(l,i)}$ exp^{i+1}	
$lexp^6$	→	`-` exp^7	
$rexp^i$	→	exp^{i+1} $qop^{(r,i)}$ ($rexp^i$ | exp^{i+1})	
exp^{10}	→	`\`$apat_1 \ldots apat_n$`->` *exp*	(λ abstraction , $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let expression)
	|	`if` *exp* `then` *exp* `else` *exp*	(conditional)
	|	`case` *exp* `of` {*alts*}	(case expression)
	|	`do` {*stmts*}	(do expression)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(function application)
aexp	→	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
	|	`(` *exp* `)`	(parenthesized expression)
	|	`(` exp_1`,` … `,` exp_k `)`	(tuple , k ≥ 2)
	|	`[` exp_1`,` … `,` exp_k `]`	(list , k ≥ 1)
	|	`[` exp_1`[,` exp_2`]` `..` `[` exp_3`]` `]`	(arithmetic sequence)
	|	`[` *exp* `|` $qual_1, \ldots, qual_n$ `]`	(list comprehension , n ≥ 1)
	|	`(` exp^{i+1} $qop^{(a,i)}$ `)`	(left section)
	|	`(` $lexp^i$ $qop^{(l,i)}$ `)`	(left section)
	|	`(` $qop^{(a,i)}_{\langle\cdot\rangle}$ exp^{i+1} `)`	(right section)
	|	`(` $qop^{(r,i)}_{\langle\cdot\rangle}$ $rexp^i$ `)`	(right section)
	|	*qcon* `{` $fbind_1, \ldots, fbind_n$ `}`	(labeled construction , n ≥ 0)
	|	$aexp_{\langle qcon\rangle}$ `{` $fbind1, \ldots, fbindn$ `}`	(labeled update , n ≥ 1)

Abstract Syntax

$48 / y / x$ $\qquad$ $(48 / y) / x$

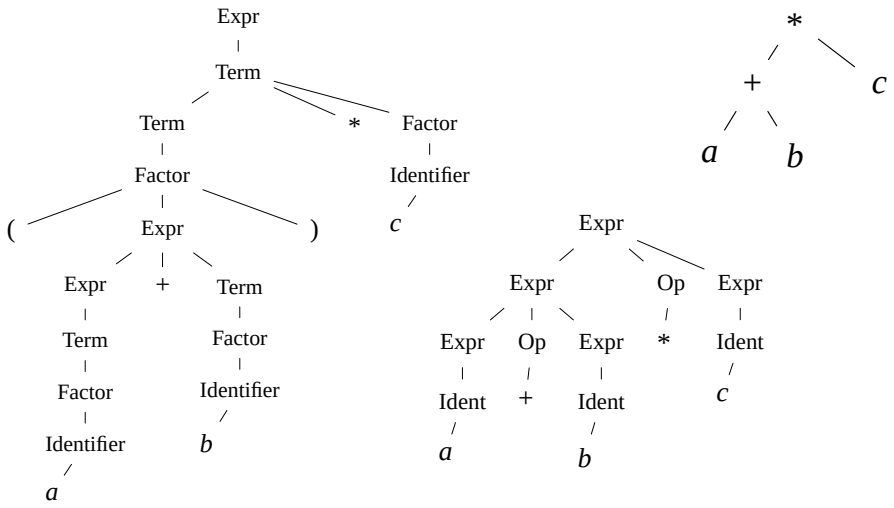
- **different** derivations in expression grammar
- **different** parse trees
- **"essentially the same"** expression
- after parsing, the **same datastructure** in the compiler:
- identical **abstract syntax trees**

Abstract syntax trees

- are, in a certain sense, "shortcuts" of the concrete syntax trees
- contain only the semantically relevant part of the program structure
- serve as bridge between syntax and semantics
  - in language definitions
  - in compilers

From Concrete to Abstract Syntax

Expr = *Ident* | *Number* | *Expr Op Expr*



Parsing and Abstract Syntax

- Parsers are based on the **concrete syntax**
- Parsers **accept** token strings as defined by the **concrete grammar**
- Parsers **implicitly** construct a derivation tree for the **concrete grammar**

- Parsers are usually directed to construct **abstract syntax trees**
- Abstract syntax trees **represent the program for compilation**
- Abstract syntax trees represent the program for **all semantical purposes**
- When comparing programming languages:

Which constructs are available in the abstract syntax?

Abstract Syntax Trees in the Textbook

Conventional representation:

- Operators as node labels: "$*$", "if_then_else_", sequence (;), "while_do_"
- Immediate constituents as successors

In the textbook:

- oriented towards implementation in Java
- Objects (records) as nodes
- Edges:
  - ***superclass reference – subclass object*** for alternatives
  - ***object – field*** for constituents

Abstract Syntax Example (Textbook / Java)

Expression = *Variable* | *Value* | *Binary*
Binary = *Operator* op; *Expression* term1, term2

```
abstract class Expression {
  // method declarations
}
class Variable extends Expression {
  String _name;
  // method implementations
}
class Binary extends Expression {
  Operator op;
  Expression term1, term2;
  // method implementations
}
```


Recursive Descent Parsing in Java

$$Expr \rightarrow Term \{ [\texttt{+} \mid \texttt{-}] Term \}^*$$

Precondition for recursive descent parsers: next unconsumed token in `token`.
Function call `input.nextToken()` prepares next token.

```
private Expression expression() {
    Binary b; Expression e;
    e = term();
    while (token.value.equals("+") || token.value.equals("-")) {
        b = new Binary();
        b.term1 = e;
        b.op = new Operator(token.value);
        token = input.nextToken();
        b.term2 = term();
        e = b;
    }
    return e;
}
```


Writing a Recursive Descent Parser from EBNF

For each non-terminal A and set of rules $A \rightarrow \omega$:

- Add a new method definition with A as return type
- Create a new object $x : A$
- For each member y of the sentential form ω,
  - If y is a non-terminal, call the method for y and assign result to appropriate field in x
  - If y is terminal, check that $token = y$ and, if so, call *nextToken*, otherwise syntax error
- If ω contains an iteration "$*$", insert a **while** loop
- If there is more than one alternative for A, insert **if** or **switch** statements that distinguish the alternatives
- Return x

Recursive Descent Parser: Textbook Class Design

- a *ConcreteSyntax* object encapsulates a *TokenStream*
- **class invariant**: the *token* field contains an unconsumed token
- most methods **parse** a syntactic category and return the abstract syntax tree

```
public class ConcreteSyntax {
    Token token;        // current token from the input stream
    TokenStream input;

  public ConcreteSyntax(TokenStream ts) { // Open the Jay source program
    input = ts;                  // as a TokenStream, and
    token = input.nextToken();   // retrieve its first Token
  }

  private void match (String s) {
    if (token.value.equals(s))  token = input.nextToken();
    else SyntaxError(s);
  }
```

Recursive Descent Parser for Jay: *program*

```
private void match (String s) {
   if (token.value.equals(s))   token = input.nextToken();
   else SyntaxError(s);
 }

 public Program program() {
   // Program --> void main ( ) '{' Declarations Statements '}'
   String[] header = {"void", "main", "(", ")"};
   Program p = new Program();
   for (int i=0; i<header.length; i++)        // bypass "void main ( )"
      match(header[i]);
   match("{");
   p.decpart = declarations();
   p.body = statements();
   match("}");
   return p;
}
```


LL(1) Grammars

Recursive descent is a **top-down LL parsing algorithm**: *left-to-right* scan of the input; *leftmost* derivation.

$$\begin{aligned} \mathit{Expr} &\rightarrow \mathit{Term}\ \{\ [\texttt{+} \mid \texttt{-}]\ \mathit{Term}\}^* \\ \mathit{Term} &\rightarrow \mathit{Factor}\ \{\ [\texttt{*} \mid \texttt{/}]\ \mathit{Factor}\}^* \\ \mathit{Factor} &\rightarrow \mathit{Ident} \mid \mathit{Integer} \mid (\mathit{Expr}) \end{aligned}$$

This is an **LL Grammar**:

- No **left-recursion** (as in e.g. $\mathit{Expr} \rightarrow \mathit{Expr}\ \texttt{+}\ \mathit{Term}$):

  $\mathit{Expr} \rightarrow \mathit{Term}\ Q \qquad Q \rightarrow \varepsilon \mid \texttt{+}\ \mathit{Expr} \mid \texttt{-}\ \mathit{Expr}$

- **Disjoint** *FIRST* **sets** for all alternatives:

  $\mathit{FIRST}(\textbf{ident}) = \{\textbf{ident}\}$

  $\mathit{FIRST}(\textbf{int}) = \{\textbf{int}\}$

  $\mathit{FIRST}(\,(\,\langle \mathsf{expr} \rangle\,)\,) = \{(\}$

Pushdown Automata

A **pushdowon automaton** (PDA) $(A, Q, q_0, F, V, v_0, \delta)$ consists of:

- a finite BI terminal alphabet A
- a finite set Q of **states**, with **initial state** $q_0 \in Q$ and **teminal states** $F \subseteq Q$
- a finite **stack alphabet** V, with an **initial stack symbol** v_0
- a finite **transistion relation** $\delta : (A^\varepsilon \times Q \times V) \leftrightarrow (Q \times V^*)$

A **configuration** is a triple in $A^* \times Q \times V^*$ consisting of unconsumed input, a state, and a stack.

The **initial configuration** for a word $w : A^*$ is $(w, q_0, \langle v_0 \rangle)$.

δ induces A-transitions and ε-transitions that always remove the top symbol from the stack and may put any number of symbols back on the stack.

An input word is **accepted** if it can lead to a configuration with

- empty stack, or
- terminal state

String Grammars — The Chomsky Hierarchy

Grammar level *Grammar type*	*Typical production*	*Recognising Automata*	*Separating Languages*
Chomsky-0 (unrestricted)	$AaQY \rightarrow QbcZd$	Turing Machines	
Chomsky-1 **context-sensitive**	$abQc \rightarrow abdYeZc$	Linearly-Bounded Automata	$a^n b^n c^n$
Chomsky-2 **context-free**	$Q \rightarrow dYeZ$	Nondet. PDA	Palindromes
		Deterministic PDA	$a^n b^n$
Chomsky-3 **regular**	$Q \rightarrow dY$	Finite-State Automata	