2 Introduction

Total 84 frames:

2.1 PL design .......................... 2
2.2 Programming paradigms ...... 3
2.3 History of programming languages .. 5
2.4 Syntax specification .......... 6
  2.4.1 Regular expressions .......... 6
  2.4.2 EBNF .................. 8
Exercises .......................... 10
2.5 Tokens: the atoms of syntax .... 10
  2.5.1 Kinds of tokens ............ 10
  2.5.2 Library identifiers ......... 13
  2.5.3 Starting point ............ 14

1. Introduction to PLs: a visual mindmap

Figure 2.1: Introduction to PLs: a visual mindmap

2.1 PL design


Universal every problem must have a solution

- Express recursive functions; it is sufficient to require
  - Conditionals
  - Loops

Natural application domain specific

Try writing a compiler in COBOL or a GUI in FORTRAN

Implementable

- Neither mathematical notation
- Nor natural language

Efficient open to debate

- More programming crimes were committed in the name of performance than for any other reason.

1Exception: domain-specific languages, e.g., pure SQL has no recursion

3. Desiderata for a PL

Expressiveness

- Turing-completeness
- But also a practical kind of expressiveness: how easy is it to program simple concepts?

Efficiency

- Recursion in functional languages is expressive but sometimes inefficient
- Is there an efficient way to implement the language (in machine code)?

Simplicity

- as few basic concepts as possible
- Sometimes a trade-off with convenience (C has “for”, who needs “while” and “do-while”?)

Uniformity and consistency of concepts

- for in PASCAL allows a single statement
- repeat...until allows any number of statements?
- Why?

Abstraction language should allow to factor out recurring patterns

Clarity to humans

The = vs. == in C is a bit confusing

Information hiding and modularity

Safety possibility to detect errors at compile time

AWK, REXX and SNOBOL type conversions are error prone

4. Guiding principles in language design

Example: The C design rules:

- Division between preprocessor, compiler and linker.
- No hidden costs
- Programmer’s accountability and responsibility

5. “Less is more”

Two program fragments to find the $n$th Fibonacci number in ALGOL-68

\[
\begin{align*}
x, y := 1; & \\
to n do (if x \lt y then x else y) := x \times y; & \\
x := \max(x, y); & \\
x, y := 1; & \\
to n do begin x := y; x := x \times y end;
\end{align*}
\]
6. A legendary Fortran bug

Computing

\[ \sum_{i=1}^{314} \sin(i) \]

with Fortran:

Fortran

\[
S = 0 \quad \text{DO 1000 I=1,314} \\
S = S + \sin(I) \\
1000 \text{ CONTINUE}
\]

But if you accidentally replace “,” by “.” the code is very different

Fortran

\[
S = 0 \quad \text{DO1000I = 1.314} \\
S = S + \sin(I) \\
1000 \text{ CONTINUE}
\]

variable \( S \) becomes simply \( \sin(1) \).

7. The Mariner 1 aborted launch

Smoke and fire launch of the Mariner 1, less than five minutes prior to its abortion by a security officer due to a combination of a hardware problem and software bug; July 22nd 1962, 09:26:16 UTC

Figure 2.2: Launch of Mariner 1

8. The bug leading to the abortion

Background:

- The “cold war” between the US and USSR
- The “space race” between the US and USSR
- America’s first planetary mission
- A very expensive project
- Designed in only 45 days
- Politicians wanted explanation
- The public wanted explanation

Outcome:

- **Official explanation:** “omission of a hyphen in coded computer instructions in the data-editing program”
- **Urban legend:** the famous Fortran bug

9. Poor design of the Fortran PL

- Bad lexical definition - spaces are immaterial
- No declaration of variables
- Implicit typing
- Poor control structure specification
- Lack of diagnostics

10. Concepts in the PL World

<table>
<thead>
<tr>
<th>Question to ask</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>How it handles values?</td>
<td>values, types and expressions</td>
</tr>
<tr>
<td>How it checks types?</td>
<td>type systems</td>
</tr>
<tr>
<td>Its entities for storing values?</td>
<td>storage</td>
</tr>
<tr>
<td>Its means for storing values?</td>
<td>commands</td>
</tr>
<tr>
<td>How it alters control?</td>
<td>sequencers</td>
</tr>
<tr>
<td>How it attaches names to values?</td>
<td>binding</td>
</tr>
<tr>
<td>How it allows generalization?</td>
<td>functions</td>
</tr>
</tbody>
</table>

Table 2.1: Concepts in the PL World

Paradigms...

2.2 Programming paradigms

Total 14 frames:

11. SQL: what’s the difference?

A query...

and a similar query...

- Think about implementing these queries in C++.
- Would the code look just the same in both cases?
12. What is a paradigm?

• paradigm (Merriam-Webster Collegiate Dictionary)
  – “a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated”
  – Model, pattern

• Thomas Kuhn (1922–1996)
  A set of “universally” recognized scientific achievements that for

• The Sapir-Whorf hypothesis: The language spoken influences the way reality is perceived.

• In PLs: a family of languages with similar basic constructs and mental model of execution

13. Main paradigms

• Imperative programming: FORTRAN, COBOL, ALGOL, PL/I, C, PASCAL, ADA, C++, ICON, MODULA-2, MODULA-3, OBERON, BASIC.

• Concurrent programming: ADA, OCCAM, X10

• Object-oriented programming: SMALLTALK, SELF, C++, OBJECTIVE-C, OBJECT-PASCAL, BETA, CLOS, EIFFEL

• Functional programming: LISP, SCHEME, MIRANDA, ML.

• Logic programming: PROLOG, PROLOG-dialects, Turbo-Prolog, ICON.

14. PL paradigms

15. The imperative paradigm

• FORTRAN, ALGOL, C, PASCAL, ADA, ...

• The program has a state reflected by storage and location

• It comprises commands (assignments, sequencers, etc.) that update the state of the program
  – They can be grouped into procedures and functions

• There are also expressions and other functional features

• Most familiar, but a large variety of possibilities must be mastered and understood

• Models real-world processes, hence still dominant

• Lends itself to efficient processing (optimizing compilers etc.)

• Will see PASCAL in recitations and home assignment

16. The functional paradigm

• LISP, SCHEME, ML, HASKELL, ...

• Everything is a function that takes arguments and returns results

• Moreover, the functions are just another kind of value that can be computed (created), passed as a parameter, etc.

• Don’t really need assignment operation or sequencers - can do everything as returning a result value of computing a function
  – E.g., use recursive activation of functions instead of iteration

• Elegant, extensible, few basic concepts

• Used for list manipulation, artificial intelligence, ...

• Requires a truly different perception using an imperative programming style in ML is even worse than a word-for-word translation among natural languages

• Will see ML, mainly in the recitations

17. Aren’t all languages pretty much the same?

• The move from C to C++ isn’t insurmountable.

• Moving from C++ to JAVA is trivial.

• And if you know JAVA, you pretty much know C#, too.

• Even if the syntax isn’t C-style used (e.g., EIFFEL), it can’t be that difficult, right?

Why make such a fuss about it?
18. The logic/declarative programming paradigm

- **PROLOG**, constraint languages, database query languages
- Predicates as the basis of execution
- Facts and rules are listed naturally
- A “computation” is *implicit* - it shows what follows from the given facts and rules
- Emphasizes what is needed, rather than how to compute it
- Used for “expert systems”
- Will see the basics of PROLOG later in the course

19. The object-oriented paradigm

- **C++**, **Smalltalk**, **Eiffel**, **Java**, **C#**
- The world has objects that contain both fields with values and operations (called methods) that manipulate the values
- Objects communicate by sending messages that ask the object to perform some method on its data
- Types of objects are declared using classes that can inherit the fields and methods of other classes
- Has become the primary paradigm because it seems to treat large systems better than other approaches
- Treated mainly in the follow-up course “Object-Oriented Programming” (236703)
- Will do a little bit of JAVA in the recitations

20. The main programming paradigms

<table>
<thead>
<tr>
<th>Imperative</th>
<th>Object Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Logical</td>
</tr>
<tr>
<td>Aspect Oriented</td>
<td>Constraints</td>
</tr>
<tr>
<td>Parallel</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: The main programming paradigms

However, there are many multi-paradigm PLs.

<table>
<thead>
<tr>
<th>Mathematica</th>
<th>OZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>F#</td>
<td><strong>Visual-Basic.Net</strong></td>
</tr>
<tr>
<td>C#</td>
<td><strong>Scala</strong></td>
</tr>
<tr>
<td><strong>Object Pascal</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Multi-paradigm PLs

2.3 History of programming languages

10 Frames:
- Writing a “Hello, World!” program in C
- Authoring “Hello, World!” with the **gvim** text editor
- Tokens are the terminals of a CFG
- What do tokens denote?
- Names aka identifiers
- Nameables
- Nameable values in C/C++/JAVA?
- Legal names
- Variations
- What’s Unicode?
- Names and kind distinction
- Names and naming conventions
- Names & your new PL
- Keywords
- Keywords & atomic types
- Atomic types in PASCAL are predefined
- Redefinition of predefined identifiers
- Reserved identifiers
- Routines whose name is a reserved identifier?
- Summary: kinds of identifiers
- The Go PL
- Why a library?
- Library identifiers
- Replaceable vs. builtin library
- Import by preprocessing
- Explicit (and implicit) import
- Implicit import
- Compilation unit
- Order of execution
- Autarkic approach
- Summary terminology

21. Language inception & evolution

Initial definition by a...

- **an Individual** LISP (**McCarthy**), APL (**Iverson**), PASCAL (**Wirth**), REXX (**Cowlishaw**), C++ (**Stroustrup**), JAVA (**Gosling**)
- **a small team** C (**Kernighan and Ritchie**), ML (**Milner et al.**), PROLOG (**Clocksin and Mellish**), ICON (**Griswold and Griswold**)
- **a committee** FORTRAN, **Algol**, PL/1, **Ada**

Some survived, many more perished for a variety of reasons

- usability
- compilation feasibility
- dependence on platform
- politics and sociology...

Most successful languages were taken over by a standards’ committees (ANSI, IEEE, ISO, ...)

22. Language genealogy (till 1990)

![Figure 2.4: Language genealogy (till 1990)](image)
23. Historical background

- Until early 1950s: no real PLs, but rather automatic programming, a mixture of assembly languages and other aids for machine code programming.
  - Mnemonic operation codes and symbolic addresses
  - Subroutine libraries where addresses of operands were changed manually
  - Interpretive systems for floating point and indexing
- Early 1950s: the Laning and Zierler System (MIT): a simple algebraic language, a library of useful functions.
- 1954: Definition of FORTRAN (FORmula TRANslator). Originally for numerical computing.
  - Symbolic expressions, subprograms with parameters, arrays, for loops, if statements, no blocks, weak control structures
  - 1957: first working compiler

24. Early 1960s:

- COBOL: Data processing. Means for data description.
- ALGOL 60: Blocks, modern control structures
  - One of the most influential imperative languages
  - Gave rise to the Algol-like languages for two decades (PASCAL, PL/I, C, ALGOL 68; SIMULA, ADA)
- LISP (list processing language): symbolic expressions (rather than numerical), computation by list manipulation, garbage collection; the first functional language

25. Mid 1960s

PL/1 an attempt to combine concepts from numerical computation languages (FORTRAN, ALGOL 60) and data processing languages (COBOL).

Simula object oriented, abstract data types


- Several OO languages: SMALLTALK, C++, EIFEL
- Logic programming: PROLOG
- Functional programming: ML, MIRANDA, HASKELL
- ADA: Another attempt, more successful than PL/1, for a general purpose language, including concurrency.
- Domain specific:
  - SNOBOL, ICON, AWK, REXX, PERL: String manipulation and scripting
  - SQL: Query language for relational databases
  - MATHEMATICA, MATLAB: Mathematical applications
  - PYTHON: large scale script programming.
  - ...

27. 1990–present

- Object oriented + WWW: JAVA, C#
- Scripting + [OO] + WWW: PERL, PYTHON, PHP, RUBY
- Client-side scripting: JAVASCRIPT
- Components and middleware between operating system and application levels
- Reuse and design patterns become useful and popular
- Multiple-language systems with standard interface - XML
- Flexibility in choice of language and moving among languages

28. Why Pascal?

- Extremely influential
- Easy to study: designed for beginners.
- Autarkic PL
- Nested functions
- Set type constructor
- Subrange types
- Nested functions and procedures
- Functions and procedures are not first class values
- Named labels, although naming is by integers

2.4 Syntax specification

19 Frames: Why a library? Library identifiers Replaceable-vs. builtin-library Import by preprocessing Explicit (and implicit) import Implicit import Compilation unit

29. Linguistics of PLs: syntax & semantics

Two components of linguistics (as in natural linguistics):

Syntax Which text files are correct programs? How expressions, commands, declarations, etc., are put together to form a program?

Semantics What’s the meaning of correct programs? Behavior when executed on a computer?

Means for specification:

Syntax Regular expressions, context-free grammars, BNF form, EBNF form, syntax diagrams (briefly touched here, subject of “Automata & formal Languages”)

Semantics Tutorials, user guides, handbooks, Wikipedia entries, language legalese (briefly touched here), and formal semantics (outside our scope).

Common theme: recursively defined sets...

2.4.1 Regular expressions

7 Frames: Order of execution Autarkic approach Summary terminology
30. Recursively defined sets

Also known as inductively defined sets

Definition 2.1 (Who is Jewish?).

- Mother Sarah was Jewish.
- Father Abraham was Jewish.
- People who converted are Jewish.
- People born to a Jewish mother are Jewish.

Other natural examples:
- Who is a Muslim?
- Who can call himself a “Dr.”?
- Who can call himself a “Rabbi”?

31. The set of strings over an alphabet

- Regular expressions is a language for defining a subset of the strings over a given alphabet.
- But, what are “strings over a given alphabet”?
  - Let \( \Sigma \) be an alphabet
  - i.e., a set of letters (also called characters), e.g., \( \Sigma = \{a, b, c, d, e, f\} \)
  - \( \Sigma \) can be used to write strings, e.g., \( \text{abba, baa, daffacc, cafe, decaf, dafca} \)
  - Let \( \varepsilon \) denote the empty string.
  - Let \( \Sigma^+ \) be the set of all strings over \( \Sigma \), length 0 string length 1 strings
  \[ \Sigma^+ = \{ \varepsilon, a, b, \ldots, f \} \] (2.1)
  - length 2 strings length 3 strings
  \[ \Sigma^+ = \{ \varepsilon, a, b, \ldots, f, a, a, b, a, a, a, \ldots \} \]

32. Recursive definition of the set of non-empty strings

Given an alphabet \( \Sigma \), the set \( \Sigma^+ \) is defined by:

**All letters** All letters in \( \Sigma \) are present in \( \Sigma^+ \)

\[ \ell \in \Sigma \Rightarrow \ell \in \Sigma^+ \] (2.2)

**Concatenation** Set \( \Sigma^+ \) is closed under concatenation:

\[ \alpha, \beta \in \Sigma^+ \Rightarrow \alpha \beta \in \Sigma^+ \] (2.3)

where \( \alpha \beta \) is the string obtained by concatenating \( \alpha \) and \( \beta \)

**Minimality** there are no other members of \( \Sigma^+ \)

\[ \gamma \in \Sigma^+ \Rightarrow \gamma \in \Sigma \ or \ \exists \alpha, \beta \in \Sigma^+ \bullet \gamma = \alpha \beta \] (2.4)

- There are many equivalent definitions of \( \Sigma^+ \)
- All these definitions must be recursive
- A string \( \alpha \in \Sigma^+ \) can be constructed in several ways by the above definition
- There is an alternative definition of \( \Sigma^+ \) by which every \( \alpha \in \Sigma^+ \) has a single, unique construction.

33. Set of strings over an alphabet—defined recursively

Definition 2.2 (Set of strings over an alphabet). Given an alphabet \( \Sigma \), the set \( \Sigma^* \) is defined by:

- \( \varepsilon \), the empty string is in \( \Sigma^* \),

\[ \varepsilon \in \Sigma^* \]

- if \( \ell \) is a letter, \( \ell \in \Sigma \), and \( \alpha \in \Sigma^* \) is a string, then

\[ \ell \alpha \in \Sigma^* \]

where \( \ell \alpha \) is the string obtained by prefixing letter \( \ell \) to string \( \alpha \)

- there are no other members of \( \Sigma^* \)

34. Examples of regular expressions

<table>
<thead>
<tr>
<th>RE</th>
<th>( S \subseteq \Sigma^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{a}</td>
</tr>
<tr>
<td>b</td>
<td>{b}</td>
</tr>
<tr>
<td>\varepsilon</td>
<td>{\varepsilon}</td>
</tr>
<tr>
<td>ab</td>
<td>{ab}</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a*</td>
<td>{\ell, \ell_2, \ell_3, \ldots}</td>
</tr>
<tr>
<td>(da</td>
<td>ba</td>
</tr>
<tr>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>

Table 2.4: Examples of regular expressions

35. Regular expressions as a recursively defined set

Given an alphabet \( \Sigma \), the set \( \text{RE}(\Sigma) \) is defined by:

**All strings** All strings in \( \Sigma^* \) are present in \( \text{RE}(\Sigma) \)

\[ \Sigma^* \subseteq \text{RE}(\Sigma) \] (2.5)

**Alternation**

\[ e_1, e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1|e_2) \in \text{RE}(\Sigma) \] (2.6)

**Concatenation**

\[ e_1, e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1e_2) \in \text{RE}(\Sigma) \] (2.7)

**Kleene closure**

\[ e \in \text{RE}(\Sigma) \Rightarrow (e^*) \in \text{RE}(\Sigma) \] (2.8)
36. Semantics can be recursively defined as well

The semantics of \( e \in \text{RE}(\Sigma^*) \) is a set \( S(e), S \subseteq \Sigma^* \)
- Strings
  \( e \in \Sigma^* \Rightarrow S(e) = \{e\} \)
- Alternation:
  \( e = (e_1 | e_2) \Rightarrow S(e) = S(e_1) \cup S(e_2) \)
- Concatenation:
  \( e = (e_1 e_2) \Rightarrow S(e) = \{\alpha \beta | \alpha \in S(e_1) \text{ and } \beta \in S(e_2)\} \)
- Kleen closure:
  \( e = (e')^* \Rightarrow S(e) = \bigcup_{i=0}^{\infty} S(e'^i) \)

37. Syntactic sugaring & other variations in regular expressions

- RE were adopted in various systems, including text editors, languages for text processing, shell scripts.
- adoptions varying syntax for the same underlying concept
- most adoptions offer syntactic sugaring

<table>
<thead>
<tr>
<th>Sugar</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-z]</td>
<td>letters a through z</td>
</tr>
<tr>
<td>[0-9]</td>
<td>all letters except digits 0 through 9</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a?</td>
<td>{a, \epsilon}</td>
</tr>
<tr>
<td>( \cup^* )</td>
<td>one or more spaces</td>
</tr>
<tr>
<td>UPPER = [A-Z]</td>
<td>name a RE to be used in the definition of other REs</td>
</tr>
</tbody>
</table>

Table 2.5: Syntax and semantics of regular expressions

- recursive use of names is never allowed.

38. Three components of a recursive definition

1. **Atoms.** e.g., the empty string is in \( \Sigma^* \)
2. **Constructors.** how to make compound members out of the atoms and compound members constructed previously.
3. **Minimality.** usually implicit, but can be phrased as
   - The set has no members other than the atoms or the compound members constructed by the construction rules.
   - The set is the intersection of all sets which are consistent with the atoms and the construction rules specification.
   - The set is the smallest set that is consistent with the atoms and the construction rules specification.

39. Recursively defined sets in PLs

- **Arithmetical expressions.**
  - **Atoms** literals, references to named entities,…
  - **Constructors** mathematical operators, user-defined functions,…
- **Executable statements (commands) in C.**
  - **Atoms** assignment, **return**,…
  - **Constructors** **if**, **for**, {…},…
- **Types in C.**
  - **Atoms** int, char,…
  - **Constructors** aka **type constructors**
    - “points to”,
    - “array of”,
    - “record with fields”, and,
    - “function taking type \( \tau \) and returning type \( \sigma \)”.
    - …

40. Example: types in Java

- Java’s types are recursively defined.
- Type constructors are e.g., class, array, and enum.
- Atomic types make the recursion base
- Atomic type are denoted in JAVA by **reserved words**:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral types</td>
<td>byte, short, int, long</td>
</tr>
<tr>
<td>Floating types</td>
<td>float, double</td>
</tr>
<tr>
<td>Other types</td>
<td>boolean, char</td>
</tr>
</tbody>
</table>

Table 2.6: Atomic types in Java

41. Compound vs. atomic members

**Atomic member** indivisible, has no components which are members

**Compound member** has smaller components which are members

Figure 2.5: A PASCAL compound command with two atomic commands in it
42. Decomposing a compound expression

Sin( (a + 3) * (a + c) )

Figure 2.6: Decomposing a compound expression

Some (but not all) of the compound expressions in the decomposition tree of the largest compound expression are marked as well.

43. Observations

- in an atomic command may contain a compound expression
- this does not make the command less atomic
- an expression never contains commands (at least not in PASCAL)
- constructors are denoted by keywords as well
- these keywords can be thought of punctuation
- or as sort of “names” of the constructors

2.4.2 EBNF

3 Frames:

44. Extended Backus-Naur form (EBNF)

<if-stmt> = if <expression> then <statement> [ else <statement> ]

- A meta-notation for describing the grammar of a language
  - Terminals = actual legal strings, written as is, or inside ”+”
  - Nonterminals = concepts of the language, written <program> or program or program in different variants
  - Rules = expanding a non-terminal to a series of NTs and Ts
- One nonterminal is designated as the start of any derivation
- A sequence of terminals not derivable from start symbol by rules of the grammar is illegal
- | is choice among several possibilities
- [...] enclose optional constructs
- a pair of “{” and “}” encloses zero or more repetitions

45. Example of an EBNF

Terminals
v n + - ( )

Nonterminals
<a> <n> <F> <E> <T>

Start Symbol
<E>

Rules
<a> = + | -
<a> = * | /
<F> = v | n
<F> = ( <E> )
<E> = <T> { <a> <T> }
<T> = <F> { <m> <F> }

46. More readable way for writing an EBNF

The common way for presenting an ENBF

- Employ meaningful names
- “Rules” is the only section; no
  - terminals list
  - non-terminals list
  - definition of a start symbol

Context free grammar for expressions

<expression> = <term> {<add-op> <term>}
<term> = <factor> {<mult-op> <factor>}
<factor> = <variable-name> | <number> | ( <expression> )
<add-op> = + | -
<mult-op> = * | /

47. Interpretation of the “expression” grammar

- Terminals never occur at the left hand side of rules: +, -, *, and /.
- Non-terminals should always occur at the left hand side rules:
  - <expression>
  - <term>
  - <factor>
- Start symbol is <expression>
- Forget about <number> and <variable-name> for now.
48. Understanding the “expression” grammar

EBNF

Context free grammar for expressions

\[
\begin{align*}
<expression> &= <term> \{<add-op> <term>\} \\
<term> &= <factor> \{<mult-op> <factor>\} \\
<factor> &= <variable-name> \\
&\quad | \ <number> \\
&\quad | \ ( <expression> ) \\
<add-op> &= + \ | \ - \\
<mult-op> &= * \ | \ / \\
\end{align*}
\]

- Is \(\frac{a + 2}{b - c} \ast 7\) a legal expression?
- Yes, because there is a sequence of rule applications, starting from \(<expression>\) that yields this string (these can be drawn as a “syntax tree”, also called “parse tree”)
- How about \(a \ast (b + c)\)?

49. Many variants for writing an EBNF

EBNFs are often written in a form intended to be readable, but only to the educated reader:

- First rule defines the start symbol
- Terminals never occur in the left
- Use more RE-like syntax for right hand of rules
- Terminals show between quotes

50. Terminals can be regular expression as well

Don’t forget \(\text{Variable-name}\) and \(\text{Number}\)

- Can potentially be specified in EBNF
- Usually have no recursion in them
- Are usually written as regular-expressions
- Are thought of as tokens or non-terminals

51. BNF vs. ENBF

Only strings of (terminals/non-terminals) can be used on the left hand side; no regular expressions in the original Backus Naur Form

Context free grammar for expressions (plain BNF)

\[
\begin{align*}
\text{Expression} &= \text{Terms} \\
\text{Terms} &= \text{Term} \text{ Addition Terms} \\
\text{Term} &= \text{Factors} \\
\text{Factors} &= \text{Factor Multiplication Factor} \\
\text{Factor} &= \text{Variable-Name} \\
&\quad | \ <number> \\
&\quad | \ ( <Expression> ) \\
\text{Addition} &= + \\
\text{Addition} &= - \\
\text{Multiplication} &= * \\
\text{Multiplication} &= / \\
\end{align*}
\]

52. Ambiguity in context free grammars

If there is a sequence of terminals with more than one derivation tree.

- Syntactical ambiguity often leads to semantical ambiguity, since there are several possible ways to “understand” the input.
- Good PL design avoids ambiguity
- It is algorithmically impossible to determine whether a BNF gives rise to ambiguity

53. Expressive power of context free grammars

Some syntactical cannot be expressed even with EBNF. Examples

- Every variable used is previously declared
- The number of arguments in a procedure call equals the number of arguments in the declaration of the procedure

Much more on grammars and identifying legal programs you will learn in the courses Automata and Formal Languages and Compilation

Exercises

1. Let \(\Sigma^*\) be the set of strings over the alphabet \(\Sigma\).
   (a) give a recursive definition for \(\Sigma^*\).
   (b) explain why there is an exponential number of strings in \(\Sigma^*\) for every string in \(\Sigma^*\)
   (c) explain why the cardinalities of \(\Sigma^{**}\) and \(\Sigma^*\) is the same.

2. Give a recursive definition for \(\Sigma^+\), the set of strings over the alphabet \(\Sigma^*\).

3. Show that
   \[
   \text{RE}(\Sigma) = \text{RE}(\text{RE}(\Sigma))
   \]

4. Give set theoretical considerations why most subsets of \(\Sigma^*\) cannot be described as regular expressions.
5. Employ set theoretical considerations to
   (a) explain why $\Sigma^\infty$, the set of all infinite strings
       over $\Sigma$, has no finite recursive definition, i.e., a
       recursive definition in which both the number of
       atoms and the number of constructors is finite.
   (b) determine whether there is such a recursive definition
       in which only the number of atoms is infinite? which
       cardinality should it be?
   (c) determine whether there is such a recursive definition
       in which only the number of constructors is
       infinite? which cardinality should it be?
   (d) Give an example of a formal language which is not
       a PL?

2.5 Tokens: the atoms of syntax

Programming involves many technical activities:

- Authoring
- Compiling
- Linking
- Executing

Concretely

```bash
rm -f hello.c a.out
cat << EOF > hello.c
#include <stdio.h>
int main(int argc, char *argv[], char **envp) {
  return printf("Hello, World!\n") <= 0;
}
EOF
c c hello.c
./a.out
```

Table 2.7: Classification of tokens in a BNF for simple expressions

Note that comments do not show up in the grammar.

Table 2.8: What do tokens denote?

Comments are not officially tokens, but they also belong
      to the atomic elements of the language.

58. Names aka identifiers

- Create an entity once, refer to it many times
- Essential for modular large-scale programming
- Largely a nuisance!
  - good names are scarce
  - difficult to make up, type, read, and understand

59. Nameables

Definition 2.3 (Nameable). A nameable is an entity kind,
such as functions, modules, types, constants, variables, for
which the programmer can provide a name.
60. Nameable values in C/C++/Java?

- Values are not in C, (the preprocessor is not part of the language)...
- not in C++, neither in JAVA,...
- but there are work-arounds:
  C/C++:

  ```c
  // Only for integer constants
  enum {
    BELL = '',
    TAB = '	',
    NL = '
',
    CR = '',
  }
  ```

  ```c
  const double E = 2.718281824;
  const int Merssene7 = 524287;
  ```

  JAVA:

  ```java
  final double E = 2.718281824;
  final int Merssene7 = 524287;
  ```

61. Legal names

All PLs include a definition of “legal names”:

**Definition 2.4 (C identifiers).** A C identifier is a series of alphanumeric characters, the first being a letter of the alphabet or an underscore, and the remaining being any letter of the alphabet, any numeric digit, or the underscore.

- Regular expression

  \ [_a-zA-Z\-2] [ _a-zA-Z0-9\-9]* \n
- Most PLs follow the same pattern.
- But, there are always annoying exceptions:
  - \TeX: digit and underscores are forbidden
  - Early Basic: a single letter, optionally followed by a digit

62. Variations

lower/UPPER case due to historical or ideological reasons.

Length limit typically 6–8 in ancient languages, ~32 or unlimited in modern languages

Special characters Can a name contain a dollar (yes, in JAVA), space (in Fortran), a quote (in JTL), or what have you?

Unicode Ain’t “α” an excellent variable name in certain contexts?

63. What’s Unicode?

- a system for encoding characters
- more than 110,000 characters
- covers ~100 scripts, representing most of the world’s writing systems
- Standard in Windows (NT/XP/Vista/2000/7), Linux, Mac OS X.
- Extends and replaces ASCII (7 bit standard, used primarily for American English)

64. Names and kind distinction

- Names can be used for very different entities.
- “Readability”

  - Prolog first character determines grammatical role (lower-case: function; uppercase/underscore: variable)
  - Perl first character determines structure, e.g., “%” for hashtable.
  - Fortran first character determines type, e.g., “i” must be an integer.

65. Names and naming conventions

Many PLs employ naming conventions,

- For distinguishing between categories, types “must” be capitalized.
- For making a single name out of multiple words:
  - **PascalCase** `FileOpen, WriteLn` (e.g., PASCAL)
  - **camelCase** `fileOpen, getClass` (e.g., JAVA)
  - **under_scoring** `file_open` (e.g., C)
  - **juxtaposition** `fileopen`, \textbackslash (e.g., \TeX)

- For denoting type, e.g., using the Hungarian Notation, denotes a variable `arru8NumberList` whose type is an array of unsigned 8-bit integers

66. Names & your new PL

Try to understand the language peculiarities:

- What “special characters” are allowed? Why?
- Is there lower/upper case distinction? Why?
- Is there a length limit? Why?
- What is the language naming convention, if any?

Remember, modern languages tend to:

1. impose no length limit
2. use Unicode
3. distinguish between upper and lower case
4. rely on conventions rather than syntax for distinguishing kinds.

\(^2\) no one knows what “readability” really means
### Keywords

**Definition 2.5** (Keywords/reserved words). A *keyword* (also called a *reserved word*) is a string of characters which makes a legal name, yet, it is reserved for special purposes and cannot be used by the programmer for any other purpose.

**Pascal Examples:**
- `program`
- `begin`
- `end`
- `record`
- `of`

### Keywords & atomic types

- Each of the atomic types in Java is denoted by a keyword.
- Some C atomic types have names made of two, and even three keywords:
  - “unsigned short”
  - “unsigned long int”
- Some C atomic types have more than one name, e.g.,
  - “long”
  - “long int”
  - “signed long”
  - “signed long int”
- Atomic types in Pascal are denoted by predefined identifiers

### Atomic types in Pascal are predefined

**Definition 2.6** (Predefined identifier). A *predefined identifier* is an identifier that is bounded to an entity (such as type, function, or value), this binding is made by the PL, with no programmer intervention, and can be bounded to another entity later on.

Redefinition of predefined identifiers is legal, but might be confusing.

### Redefinition of predefined identifiers

Confusing program

```plaintext
Program misnomer;
TYPE
Double = Real;
Boolean = Integer;
VAR
Integer: Boolean;
Real: Double;
BEGIN
Integer := 3;
Real := Integer / Integer;
WriteLn('i = ',Integer,' r = ',real)
END.
```

$i = 3$ $r = 1.00000000000000E+000$

Observe that my pretty printer got confused as well.

### Reserved identifiers

**Definition 2.7** (Reserved identifier). A *reserved identifier* is a keyword used as an identifier.

- The word `int` is a reserved identifier, it identifies the *integer* atomic type.
- The identifiers `Integer` and `WriteLn` in Pascal are not reserved. The programmer may redefine these.
- Not all reserved words are reserved identifiers:
  - `return` in C (an atomic command).
  - `begin`, `end`, `program` in Pascal (punctuation).
  - `struct` in C (a constructor for creating compound types from other types)

### Routines whose name is a reserved identifier?

In most PLs, the names of “standard” routines (procedures and functions) are not reserved:

- They are either imported or builtin.
- A notable exception is AWK:
  - `print` A builtin function, for printing.
  - `exit` A builtin function, stopping execution.
  - `int` A builtin function, for conversion into an integral type.

Unlike Pascal, builtin names in AWK are reserved.

### Summary: kinds of identifiers

- Identifiers:
  - Reserved identifier
  - Predefined identifier
  - Library identifier
  - Other

In addition, we have those reserved words which are not identifiers.
- Identifiers reserved for future use
Denotation of atomic entities
- Punctuation (often used in constructors of entities)
- Other, e.g., marking Boolean attributes (\texttt{register, auto, static} in the C PL)
- there are so many PLs, we cannot hope to classify them all

### 74. The Go PL
Can you classify the identifiers and reserved words used here?

```go
// "Hello, World!" in Go
package main
import "fmt"
func main() { // main function
    fmt.Printf("Hello, World!\n")
}
```

- \texttt{package} reserved word, punctuation
- \texttt{import} reserved word, punctuation
- \texttt{func} reserved word, punctuation
- \texttt{main} identifier, other
- \texttt{fmt} identifier, library
- \texttt{Printf} identifier, library

#### 2.5.2 Library identifiers

**0 Frames:**

1. **Why a library?**
   - The set of executable commands is always a recursively defined set.
   - Derivation rules are language dependent, typically including blocks, iterations, conditionals, and routines
   - Atomic executables include
     - Commands denoted by keywords, e.g., \texttt{return, break} and \texttt{continue}
     - Other atomic commands such as \texttt{assignment}.
   - Invocation of routines

   Some routines are so
   - low-level that they cannot be implemented within the language
   - essential that there is little point in having each programmer redo them
   - tiresome that most programmers could not be bothered implementing them

#### 76. Library identifiers

**Definition 2.8** (Library). A collection of pre-made routines (or modules) that are available to the programmer.

- **standard library** replaceable (as in C)
- **builtin library** cannot be replaced (as in PASCAL and AWK)

Identification of entities in the library
- **Reserved words** rare (e.g., AWK)
- **Pre-defined identifiers** as in PASCAL.
- **Importing** as in JAVA and C.

#### 77. Replaceable- vs. builtin- library

<table>
<thead>
<tr>
<th>Replaceable</th>
<th>Builtin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Troublesome for programmer</td>
<td>Less work for the programmer</td>
</tr>
<tr>
<td>Small language specification</td>
<td>Bulky language specification</td>
</tr>
<tr>
<td>Flexible</td>
<td>Rigid</td>
</tr>
<tr>
<td>Modular language design</td>
<td>Tangled language design</td>
</tr>
<tr>
<td>Library can be very large</td>
<td>Library is typically small</td>
</tr>
<tr>
<td>Most modern PLs</td>
<td>PLs designed for beginners and for one-liner/scripting</td>
</tr>
</tbody>
</table>

**Table 2.9: Replaceable- vs. builtin- library**

- **Dinosaurs**: Languages such as COBOL which included huge builtin library tend to collapse under their own weight

#### 78. Import by pre-processing

Import at the source, textual level

```c
#include <stdio.h>
int main(int argc, char *argv[], char **envp) {
    return printf("Hello, World!\n") <= 0;
}
```

- \texttt{$\text{include}$ <stdio.h>}
- \texttt{int main(int argc, char *argv[], char **envp) {
  return printf("Hello, World!\n") <= 0;
  }}

#### 79. Explicit (and implicit) import

Your program declares which library identifiers it uses:
- The keyword \texttt{import} seems to be used in so many PLs.
- Other languages may use other keywords, e.g., \texttt{uses}

Pre-Processor
Semantics is greater than textual import.

**Properties of import**

- usually carried out for a bunch of identifiers (for now, we call such a bunch a module)
- there is an implicit search path for the library
- may be used also for user-provided (non-library) modules
- may cause other modules to compile

**80. Implicit import**

Implicit Import

Certain principal modules are automatically imported even if the programmer does not explicitly import these

e.g., `java.lang.*` in JAVA

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

**81. Compilation unit**

Definition 2.9 (Compilation unit). compilation unit is a portion of a computer program which is sufficiently complete to be compiled correctly.

- Usually a file
- Can be string, or a buffer of the editor

**2.5.3 Starting point**

Yet, another thing to observe in any “Hello, World!” program:

- Normally sequential
- Can be changed by
  - Conditional commands
  - Iteration commands
  - Parallelization commands
  - Invoking routines
  - ...

*But, in the presence of several compilation units, or even several routines in the same compilation unit, where do we start?*

**83. Autarkic approach**

**au-tar-ky or au-tar-chy, pl. au-tar-kies or au-tar-phies**

1. A policy of national self-sufficiency and nonreliance on imports or economic aid.
2. A self-sufficient region or country.
List of Equations

(2.1) $\Sigma^*$, the set of all strings over the alphabet $\Sigma = \{a,b,c,d,e,f\}$ ................................. 6

- length 0 string
- length 1 strings
  $\Sigma^* = \{ \varepsilon, a,b,...,f \}$
- length 2 strings
  $\Sigma^* = \{ aa,ab,...,ff \}$
- length 3 strings
  $\Sigma^* = \{ aaa,...,fff \}$

(2.2) Atoms of $\Sigma^+$ ........................................ 7

$\ell \in \Sigma \Rightarrow \ell \in \Sigma^+$

(2.3) Concatenation constructor of $\Sigma^+$ ......... 7

$\alpha,\beta \in \Sigma^+ \Rightarrow \alpha\beta \in \Sigma^+$

(2.4) All members of $\Sigma^+$ are either atomic strings, or constructed by concatenation from other strings in $\Sigma^+$ 7

$\gamma \in \Sigma^+ \Rightarrow \gamma \in \Sigma$ or $\exists \alpha, \beta \in \Sigma^+ \bullet \gamma = \alpha\beta$

(2.5) All strings can be thought of as regular expressions ................................. 7

$\Sigma^* \subseteq \text{RE}(\Sigma)$

(2.6) Regular expressions constructor I: alternation 7

$e_1,e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1|e_2) \in \text{RE}(\Sigma)$

(2.7) Regular expressions constructor II: concatenation 7

$e_1,e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1e_2) \in \text{RE}(\Sigma)$

(2.8) Regular expressions constructor III: Kleene closure 7

$e \in \text{RE}(\Sigma) \Rightarrow (e^*) \in \text{RE}(\Sigma)$

(2.9) Regular expression defining identifiers' syntax in a typical modern PL 11

$[_a-zA-Z][_a-zA-Z0-9]^*$