Section 2

Introduction

1. Preliminaries

2. Introduction

2.1 PL design
2.2 Programming paradigms
2.3 History of programming languages
2.4 Syntax specification
2.5 Tokens: the atoms of syntax
2. Introduction

2.1. PL design
Requirements from a PL

**Universal** every problem must have a solution (Exception: domain-specific languages, e.g., pure SQL has no recursion)
- 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.
Desiderata for a PL I

**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?
Desiderata for a PL II

Software Engineering

- **Abstraction** – hide details
- **Modularity** – constrain dependences

Language Mechanism

- **Information hiding** and **encapsulation**
  Language should allow to factor out recurring patterns

Clarity to humans

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

Safety

Possibility to detect errors at compile time

AWK, Rexx and Snobol type conversions are error prone
Guiding principles in language design

Example: The C design rules:

- Division between preprocessor, compiler and linker.
- No hidden costs
- Programmer’s accountability and responsibility
“Less is more”

Two program fragments to find the $n^{th}$ Fibonacci number in Algol-68

```
x, y := 1;
to n do (if x < y then x else y) := x + y;
x := max(x, y);
```

```
x, y := 1;
to n do begin x, y := y, x; x := x + y end;
```
A legendary **FORTRAN** bug

Computing

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

with **FORTRAN**:

```
S = 0
DO 1000 I=1,314
S = S + SIN(I)
1000 CONTINUE
```

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

```
S = 0
DO1000I = 1.314
S = S + SIN(I)
1000 CONTINUE
```

variable \( S \) becomes simply \( \sin(1) \).
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 22\textsuperscript{nd} 1962, 09:26:16 UTC
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
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

```fortran
S = 0
DO 1000 I=1.314
T = T + SIN(I)
1000 CONTINUE
```
# Concepts in the PL World

The table below lists some key questions about programming languages (PLs) and the concepts that characterize them:

<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>

Paradigms...
2. Introduction

2.2. Programming paradigms

1. Preliminaries

2. Introduction
SQL: what’s the difference?

A query,…

```
SELECT firstName, lastName, age
FROM user
WHERE firstName = "David"
ORDER BY age
```

and a similar query…

```
SELECT firstName, lastName, age
FROM user
WHERE age > 18 AND age < 65
ORDER BY age
```

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

- par-a·digm (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
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.
(Not comprehensive)
(Not to scale)
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
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
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?
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
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
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>

However, there are many multiparadigm PLs.

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

2.3. History of programming languages
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, …)
Language genealogy (till 1990)
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
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/1, C, **Algol** 68; **Simula**, **Ada**)
- **LISP** (list processing language): symbolic expressions (rather than numerical), computation by list manipulation, garbage collection; *the first functional language*
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
1970–1990

- Several OO languages: SMALLTALK, C++, EIFFEL
- Logic programming: PROLOG
- Functional programming: ML, MIRANDA, HASKELL
- ADA: Another attempt, more successful than PL/I, 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.
  - ...
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**
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. Introduction

2.4. Syntax specification

1. Preliminaries

2. Introduction

2.1 PL design

2.2 Programming paradigms
2.3 History of programming languages
2.4 Syntax specification
2.4.1 Regular expressions
2.4.2 EBNF
2.5 Tokens: the atoms of syntax
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. Introduction

2.4. Syntax specification

2.4.1. Regular expressions
Recursively defined sets

Also known as *inductively defined sets*

**Definition (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”?
2. Introduction

2.4. Syntax specification / 2.4.1. Regular expressions

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., abba, baa, daffacc, cafe, decaf, dafcaa
  - Let $\varepsilon$ denote the empty string.
  - Let $\Sigma^*$ be the set of all strings over $\Sigma$,
    \[ \Sigma^* = \{\varepsilon, a, b, \ldots, f, \text{length 0 string}, \text{length 1 strings}, \text{length 2 strings}, \text{length 3 strings}, \text{aaa, ab, \ldots, ff, aaa, \ldots, fff, \ldots}\} \]
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 $\Sigma^+$

$$\ell \in \Sigma \Rightarrow \ell \in \Sigma^+ \quad (4.2)$$

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

$$\alpha, \beta \in \Sigma^+ \Rightarrow \alpha \beta \in \Sigma^+ \quad (4.3)$$

where $\alpha \beta$ is the string obtained by concatenating $\alpha$ and $\beta$

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

$$\gamma \in \Sigma^+ \Rightarrow \gamma \in \Sigma \text{ or } \exists \alpha, \beta \in \Sigma^+ \bullet \gamma = \alpha \beta \quad (4.4)$$
Other recursive definitions of the set of strings

- 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.
Set of strings over an alphabet—defined recursively

Definition (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 $\Sigma^*$
## 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>${\varepsilon, a, aa, aaaa, \ldots}$</td>
</tr>
<tr>
<td>$(da</td>
<td>ba) a^* (d</td>
</tr>
<tr>
<td>$(a</td>
<td>b</td>
</tr>
</tbody>
</table>
Regular expressions as a recursively defined set

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

- All strings
  \[ \Sigma^* \subseteq \text{RE}(\Sigma) \]  \hspace{1cm} (4.5)

- Alternation
  \[ e_1, e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1|e_2) \in \text{RE}(\Sigma) \]  \hspace{1cm} (4.6)

- Concatenation
  \[ e_1, e_2 \in \text{RE}(\Sigma) \Rightarrow (e_1e_2) \in \text{RE}(\Sigma) \]  \hspace{1cm} (4.7)

- Kleene closure
  \[ e \in \text{RE}(\Sigma) \Rightarrow (e^*) \in \text{RE}(\Sigma) \]  \hspace{1cm} (4.8)
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:**

  
  
  $e = (e_1e_2) \Rightarrow S(e) = \{\alpha\beta \mid \alpha \in S(e_1) \text{ and } \beta \in S(e_2)\}$

- **Kleene closure:**

  
  
  $e = (e')^* \Rightarrow S(e) = \bigcup_{i=0}^{\infty} S\left(\overbrace{e' \cdots e'}^{i \text{ times}}\right)$
Syntactic sugaring & other variations in regular expression

- 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>Meaning</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-z]</td>
<td>inclusive range range</td>
<td>letters a through z</td>
</tr>
<tr>
<td>[^0-9]</td>
<td>exclusive range</td>
<td>all characters in the alphabet, except digits 0 through 9</td>
</tr>
<tr>
<td>a?</td>
<td>optional</td>
<td>{a, \epsilon}</td>
</tr>
<tr>
<td>\s+</td>
<td>one or more</td>
<td>one or more spaces</td>
</tr>
<tr>
<td>UPPER = [A-Z]</td>
<td>naming</td>
<td>name a RE to be used in the definition of other REs</td>
</tr>
<tr>
<td>${UPPER}[a-z]</td>
<td>using name</td>
<td>upper case letter followed by a lower case letter</td>
</tr>
</tbody>
</table>

Note: recursive use of names is never allowed.
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.
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 $\sigma$”.
    - …
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>
Compound vs. atomic members

Atomic member  indivisible, has no components which are members
Compound member  has smaller components which are members

Begin
    a := b * c
    ;
    WriteLn( Sin( (a + 3) * (a + c) ) )
end
Decomposing a compound expression

\[
\sin( (a + 3) * (a + c) )
\]

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

2.4. Syntax specification

2.4.2. *EBNF*
Extended Backus-Naur form (EBNF)

\[ <\text{if-stmt}> = \text{if} \ <\text{expression}> \ \text{then} \ <\text{statement}> \ [ \ \text{else} \ <\text{statement}> \ ] \]

- 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 \text{<program>} or \text{program} or \text{program} in different variants
  - Rules = expanding a non-terminal to a series of nonterminals 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
- \mid is choice among several possibilities
- \[ \ldots \] enclose optional constructs
- A pair of “{” and “}” encloses zero or more repetitions
Example of an EBNF

Terminals
v n + - ( )

Nonterminals
<a> <m> <F> <E> <T>

Start Symbol
<E>

Rules
<a> = + | -
<m> = * | /
<F> = v | n
<F> = ( <E> )
<E> = <T> {<a> <T>}
<T> = <F> {<m> <F>}
More readable way for writing an EBNF

The common way for presenting an EBNF

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

Context free grammar for expressions

\[
\begin{align*}
\langle \text{expression} \rangle &= \langle \text{term} \rangle \ {\langle \text{add-op} \ \langle \text{term} \rangle \rangle} \\
\langle \text{term} \rangle &= \langle \text{factor} \rangle \ {\langle \text{mult-op} \ \langle \text{factor} \rangle \rangle} \\
\langle \text{factor} \rangle &= \langle \text{variable-name} \rangle \\
&\quad \mid \langle \text{number} \rangle \\
&\quad \mid ( \langle \text{expression} \rangle ) \\
\langle \text{add-op} \rangle &= + \mid - \\
\langle \text{mult-op} \rangle &= * \mid / 
\end{align*}
\]
Interpretation of the “expression” grammar

Context free grammar for expressions

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

- 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.
Understanding the “expression” grammar

Context free grammar for expressions

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

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

- How about \(a \times (b + c)\)?
Many variants for writing an EBNF

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

Context free grammar for expressions (another EBNF syntactical variant)

Expression ::= Term (('+' | '-') Term)*
Term ::= Factor (('*' | '/') Factor )* 
Factor ::= Variable-Name 
| Number 
| '(' Expression ')' 

- 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
Terminals can be regular expression as well

Don’t forget **Variable-name** and **Number**
- Can potentially be specified in EBNF
- Usually have no recursion in them
- Are usually written as regular-expression
- Are though of as *tokens* or *non-terminals*

**Context free grammar for expressions with regular expressions for tokens**

Expression ::= Term (('+' | '-') Term)*
Term ::= Factor (('*' | '/') Factor )*
Factor ::= Variable-Name
  | Number
  | '(' Expression ')'
Variable-Name := [a-zA-Z][a-zA-Z0-9]*
Number := [+-]? [0-9]+
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)

Expression = Terms
Terms: Term
Terms: Term Addition Terms;
Term: Factors;
Factors: Factor:
Factors: Factor Multiplication Factor;
Factor: Variable-Name;
Factor: Number;
Factor: '(' Expression ')';
Addition: '+';
Addition: '-';
Multiplication: '*';
Multiplication: '/';
Ambiguity in context free grammars

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

- Syntactical ambiguity often leads to ambiguous semantics, since there are several possible ways to interpret the input.
- Good PL design avoids ambiguity
- It is algorithmically impossible to determine whether a BNF gives rise to ambiguity
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*
2. Introduction

2.5. Tokens: the atoms of syntax

1. Preliminaries

2. Introduction

2.1 PL design

2.2 Programming paradigms

2.3 History of programming languages

2.4 Syntax specification

2.5 Tokens: the atoms of syntax

2.5.1 Kinds of tokens

2.5.2 Library identifiers

2.5.3 Starting point
Writing a “Hello, World!” program in C

Using Bash, Gnu/Linux, etc.

Programming involves many technical activities:

Actions

- Authoring
- Compiling
- Linking
- Executing

Concretely

```
rm -f hello.c a.out

cat << EOF > hello.c
#include <stdio.h>

int main(int argc, char *argv[], char *envp)
{
    return printf("Hello, \nWorld!\n") <= EOF
}
EOF

c -l hello.c
./a.out
```
The **gtksourceview** library makes these so similar...

Still, the similarity of PLs makes **gtksourceview** possible!
2. Introduction

2.5. Tokens: the atoms of syntax

2.5.1. Kinds of tokens
Tokens are the terminals of a CFG

Context free grammar for expressions (plain BNF)

Expression = Terms
Terms: Term
Terms: Term Addition Terms;
Term: Factors;
Factors: Factor:
Factors: Factor Multiplication Factor;
Factor: Variable-Name;
Factor: Number;
Factor: '(' Expression ')';
Addition: '+';
Addition: '-';
Multiplication: '*';
Multiplication: '/';

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;(&quot;, &quot;)&quot;</td>
<td>punctuation</td>
</tr>
<tr>
<td>&quot;*&quot;, &quot;+&quot;</td>
<td>operators</td>
</tr>
<tr>
<td>Variable-Name</td>
<td>identifier</td>
</tr>
<tr>
<td>Number</td>
<td>literal</td>
</tr>
</tbody>
</table>

Note that comments do not show up in the grammar.
What do tokens denote?

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
<th>Denotes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td><code>main, i, printf, argv</code></td>
<td>a “nameable”</td>
</tr>
<tr>
<td>Literal</td>
<td>“132”,</td>
<td>itself</td>
</tr>
<tr>
<td></td>
<td>&quot;Hello, World\n&quot;</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>“<em>, ‘+’, ‘</em>’, ‘/’”</td>
<td>a builtin function</td>
</tr>
<tr>
<td>Punctuation</td>
<td>“;’, ‘,’”, ‘(’, ‘)’</td>
<td>nothing (reading and parsing aide)</td>
</tr>
<tr>
<td>Reserved word</td>
<td><code>if, class, int</code></td>
<td>...</td>
</tr>
<tr>
<td>Reserved identifier</td>
<td><code>int, class, int</code></td>
<td>primitive type</td>
</tr>
<tr>
<td>Predefined identifier</td>
<td>Integer, &amp;primitive type</td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td>/* fubar */</td>
<td>Nothing!</td>
</tr>
</tbody>
</table>

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

2.5. Tokens: the atoms of syntax

2.5.1. Kinds of tokens

Nameables

Definition (Nameable)

A nameable is an entity kind, such as functions, modules, types, constants, variables, for which the programmer can provide a name.

Nameable values in Pascal

```
CONST
Pi = 3.14159265358979323846264
```

Nameable types in C

```
// Type named struct Date:
struct Date {
    int month, day, year;
};
```
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 workarounds:

C/C++:

```c
// Only for integer constants
enum {
    BELL = '\b',
    TAB = '\t',
    NL = '\n',
    CR = '\r',
}
```

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

JAVA:

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

All PLs include a definition of “legal names”:

Definition (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

\[ [\_a-zA-Z] [\_a-zA-Z0-9]* \] (5.1)

- 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
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?
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)
Names and kind distinction

- Names can be used for *very different* entities.
- “Readability” (no one knows what “readability” really means”), concerns as well as parsing issues make PL impose rules such as:

  - **Prolog** first character determines *grammatical role* (lowercase: 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.
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`, `	extbackslash`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**
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.
Keywords

Definition (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

a somewhat confusing fact of life

---

**Definition (Predefined identifier)**

A *predefined identifier*

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

---

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

Confusing program

(* misnomer *)
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 = 1.0000000000000000E+000

Observe that my pretty printer got confused as well.
Reserved identifiers

**Definition (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* nor *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 (register, auto, static in the C PL)
- there are so many PLs, we cannot hope to classify them all
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")
}
```

- **package** reserved word, punctuation
- **import** reserved word, punctuation
- **func** reserved word, punctuation
- **main** identifier, other
- **fmt** identifier, library
- **Printf** identifier, library
2. Introduction

2.5. Tokens: the atoms of syntax

2.5.2. Library identifiers
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., *return*, *break* and *continue*
  - Other atomic commands such as *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
Library identifiers

**Definition (Library)**

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

Standard library replaceable (as in C)

Built-in library cannot be replaced (as in PASCAL and AWK)

Identification of entities in the library

Reserved words rare (e.g., AWK)

Predefined identifiers as in PASCAL

Importing as in JAVA and C.
## 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>

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

Import at the source, textual level

```c
#include <stdio.h>

int main(int argc, char *argv[], char **envp)
{
    return printf("Hello, World!\n") <= 0;
}
```

Preprocessor
Explicit (and implicit) import

Your program declares which library identifiers it uses:

- The keyword `import` seems to be used in so many PLs.
- Other languages may use other keywords, e.g., `uses`

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
Implicit import

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

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

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

Definition (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. Introduction

2.5. Tokens: the atoms of syntax

2.5.3. Starting point
Order of execution

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

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

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

**au·tar·ky or au·tar·chy, pl. au·tar·kies or au·tar·chies**

1. A policy of national self-sufficiency and non-reliance on imports or economic aid.
2. A self-sufficient region or country.
Summary terminology

- identifiers, nameable entities (variables, values, functions, procedures, templates, namespaces, labels, modules),
- keywords, reserved identifiers, predefined identifiers
- literals, escaping, comments.
- separatist, terminist, and variations.