Goals of today’s lecture

- Revisit control structures
- Read and understand the syntax description for Eiffel
- Write simple syntax descriptions

Control structures

**Sequence**: “To achieve C from A, first achieve an intermediate goal B from A, then achieve C from B”

**Loop**: solve the problem on successive approximations of its input set

**Conditional**: solve the problem separately on two or more subsets of its input set
Control structures

Sequence:
- instruction
- instruction
- instruction

Loop:
- from
- compound
- invariant
- expression
- variant
- expression
- until
- expression
- loop
- compound
- end

Conditional:
- if
- condition
- then
- compound
- elseif
- condition
- then
- compound
- else
- compound
- end

Intro. to Programming, lecture 9: Describing the syntax

Flowcharts

One-entry, one-exit

Intro. to Programming, lecture 9: Describing the syntax
Control structures at the machine level

Unconditional branch:

\[ \text{BR} \quad \text{label} \]

Conditional branch, for example:

\[ \text{BEQ} \quad \text{loc}_a \quad \text{loc}_b \]

The equivalent of if-then-else

\[ \text{if } a = b \text{ then } \text{Compound}_1 \text{ else } \text{Compound}_2 \text{ end} \]

In programming languages: the Goto

\[ \text{test condition goto if_part Compound_2 goto continue} \]

\[ \text{if_part: Compound}_1 \]

\[ \text{continue: } \text{... Continuation of program ...} \]
“Goto considered harmful”

Arbitrary Goto instructions lead to messy, hard to maintain programs (“spaghetti code”)

The Goto today

Almost universally decried
Still exists in some programming languages
Also hides under various disguises, e.g. break

```
  loop
  ...
  if c then break end
  ...
  end
```

Control structures

<table>
<thead>
<tr>
<th>Sequence: instruction instruction instruction</th>
</tr>
</thead>
</table>

| Loop: from compound invariant expression variant expression until expression loop compound end |

| Conditional: if condition then compound elseif condition then compound else compound end |
Syntax: Conditional

A conditional instruction consists, in order, of:
An "If part", of the form if condition.
A "Then part" of the form then compound.
Zero or more "Else if parts", each of the form elseif condition then compound.
Zero or one "Else part" of the form else compound
The keyword end.

Here each condition is a boolean expression, and each compound is a compound instruction.

The power of human mind

I cdnoult bivelee taht I cluod aulacity uesdnatrnd waht I was rdgnie. The Paomnnehal Pwear of the Hmuan Mnid Aoccdrng to a rscheearch at Cmabrigde Uinervtisy, is deosn’t mtaer in what oredr the ltteers in a wrod are, the olny iprmntnt tihng is taht the frist and lsat ltteer be in the rghit pclae. The rset can be a taotl mses and you can stil rd it wouthit prbelm. Tht is bcuseae the huamn mnid deos not rd evry ltteer by istlef, but the wrod as a wlohe. Pprety Amzanig Huh?

Why describe the syntax formally?

We know syntax descriptions for human languages:

- e.g. grammar for German, French, ...
- Expressed in natural language
- Good enough for human use
- Ambiguous, like human languages themselves
Why describe the syntax formally?

Programming languages need better descriptions:
- More precise: must tell us unambiguously whether given program text is legal or not
- Use formalism similar to mathematics
- Can be fed into compilers for automatic processing of programs

Compilers use algorithms to
- Determine if input is correct program text (parser)
- Analyze program text to extract specimens
- Translate program text to machine instructions

Compilers need strict formal definition of programming language

Formal Description of Syntax

Use formal language to describe programming languages.

Languages used to describe other languages are called Meta-Languages

Meta-Language used to describe Eiffel:
- BNF-E (Variant of the Backus-Naur-Form, BNF)
### History

1954 FORTRAN: First widely recognized programming language (developed by John Backus et al.)

1958 ALGOL 58: Joint work of European and American groups

1960 ALGOL 60: Preparation showed a need for a formal description → John Backus (member of ALGOL team) proposed Backus-Normal-Form (BNF)

1964: Donald Knuth suggested acknowledging Peter Naur for his contribution → Backus-Naur-Form

Many variants since then, e.g. graphical variant by Niklaus Wirth

### Formal description of a language

BNF lets us describe syntactical properties of a language

Remember: Description of a programming language also includes lexical and semantic properties → other tools

### Formal Description of Syntax

A language is a set of phrases

A phrase is a finite sequence of tokens from a certain "vocabulary"

Not every possible sequence is a phrase of the language

A grammar specifies which sequences are phrases and which are not

BNF is used to define a grammar for a programming language
Example of phrases

```plaintext
class PERSON
  feature
    age: INTEGER
      -- Age
  end
end PERSON
```

Grammar

Definition

A Grammar for a language is a finite set of rules for producing phrases, such that:

1. Any sequence obtained by a finite number of applications of rules from the grammar is a phrase of the language.

2. Any phrase of the language can be obtained by a finite number of applications of rules from the grammar.

Elements of a grammar: Terminals

Terminals

Tokens of the language that are not defined by a production of the grammar. E.g. keywords from Eiffel such as if, then, end or symbols such as the semicolon ";" or the assignment ":="
Elements of a grammar: Nonterminals

Nonterminals

Names of syntactical structures or substructures used to build phrases.

Elements of a grammar: Productions

Productions

Rules that define nonterminals of the grammar using a combination of terminals and (other) nonterminals.

An example production

Conditional:

```
if Condition then Instruction
else Instruction
end
```
BNF Elements: Concatenation

Graphical representation:

```
A ----> B
```

BNF: \[ A \ B \]
Meaning: \( A \) followed by \( B \)

---

BNF Elements: Optional

Graphical representation:

```
A
```

BNF: \[ [ \ A \ ] \]
Meaning: \( A \) or nothing

---

BNF Elements: Choice

Graphical representation:

```
A

\|\n
B
```

BNF: \[ A \ | \ B \]
Meaning: either \( A \) or \( B \)
**BNF Elements: Repetition**

**Graphical representation:**

```
A
```

**BNF:** \( A \)\

**Meaning:** sequence of zero or more A

---

**BNF Elements: Repetition, once or more**

**Graphical representation:**

```
A
```

**BNF:** \( A \)\

**Meaning:** sequence of one or more A

---

**BNF elements: Overview**

**Concatenation:** A B

```
A -- B
```

**Optional:** [ A ]

```
A
```

**Choice:** A | B

```
A
```
```
B
```

**Repetition (zero or more):** \( A \)\

```
A
```

**Repetition (at least once):** \( A \)\

```
A
```
A simple example

Example phrases:
- 76
- 1.56
- 12.845
- -1.34
- 13.0

Translate it to written form!

Intro. to Programming, lecture 9: Describing the syntax

BNF Elements Combined

written in BNF:

```
digit  △ 0 1 2 3 4 5 6 7 8 9
float_number  △  [ ] { digit }* { digit }+
```

```
Conditional:
```

written in BNF:
```
Conditional:

```
**BNF: Conditional with elseif**

Conditional \( \triangleq \) (Then_part_list [Else_part] end)

Then_part_list \( \triangleq \) Then_part \{ (elseif Then_part) \}*

Then_part \( \triangleq \) Boolean_expression then Compound

Else_part \( \triangleq \) elseif Compound

**Different Grammar for Conditional**

Conditional \( \triangleq \) If_part Then_part Else_list end

If_part \( \triangleq \) if Boolean_expression

Then_part \( \triangleq \) then Compound

Else_list \( \triangleq \) { elseif_part }* [ else Compound ]

Elseif_part \( \triangleq \) elseif Boolean_expression Then_part

**Simple BNF example**

Sentence \( \triangleq \) I [ don't ] Verb Names Quant

Names \( \triangleq \) Name (and Name)*

Name \( \triangleq \) tomatoes | shoes | books | football

Verb \( \triangleq \) like | hate

Quant \( \triangleq \) a lot | a little

Which of the following phrases are correct?

- I like tomatoes and football
- I don't like tomatoes a little
- I hate football a lot
- I like shoes and tomatoes a little
- I don't hate tomatoes, football and books a lot

Rewrite the BNF to include the incorrect phrases
Simple BNF example (Solution)

Which of the following phrases are correct?
- I like tomatoes and football
- I don't like tomatoes a little
- I hate football a lot
- I like shoes and tomatoes a little
- I don't hate tomatoes, football and books a lot

Rewrite the BNF to include the incorrect phrases
Sentence △ I [don't] Verb Names [Quant]
Names △ Name [[, Name]* and Name]
Name △ tomatoes | shoes | books | football
Verb△ like | hate
Quant △ a lot | a little

BNF-E

Used in official description of Eiffel.
Every Production is one of

Concatenation
A △ B C [ D ]

Choice
A △ B | C | D

Repetition
A △ { B terminal ... }*
(Also with +)

BNF-E Rules

- Every nonterminal must appear on the left-hand side of exactly one production, called its defining production
- Every production must be of one kind: Concatenation, Choice or Repetition
Conditional with **elseif** (BNF)

```
Conditional \[ If \]
  Then_part_list \[ If \] [Else_part] end
Then_part_list \[ If \] \[ Else \] \{ elseif \[ If \] Then_part \[ If \] \}^*
  Then_part \[ If \] Boolean_expression \[ If \] then Compound
   Else_part \[ If \] else Compound
```

**BNF-E: Conditional**

```
Conditional \[ If \]
  Then_part_list \[ If \] [Else_part] end
Then_part_list \[ If \] \{ \[ Else \] \}^*
  Then_part \[ If \] Boolean_expression \[ If \] then Compound
   Else_part \[ If \] else Compound
```

**Recursive grammars**

- Constructs may be nested
- Express this in BNF with **recursive grammars**
- Recursion: circular dependency of productions
Recursive grammars

Conditionals can be nested within conditionals:

\[
\text{Else_part} \triangleq \text{else Compound}
\]

\[
\text{Compound} \triangleq \{ \text{Instruction} \}^*
\]

\[
\text{Instruction} \triangleq \text{Conditional} \mid \text{Loop} \mid \text{Call} \mid ...
\]

Recursive grammars

Production name can be used in its own definition

Definition of Then_part_list with repetition:

\[
\text{Then_part_list} \triangleq \{ \text{Then_part} \text{ elseif} \}^*
\]

Recursive definition of Then_part_list:

\[
\text{Then_part_list} \triangleq \text{Then_part} \{ \text{elseif} \text{ Then_part_list} \}
\]

Conditional

\[
\text{if } a = b \text{ then}
\]

\[
\text{a} := a - 1
\]

\[
\text{b} := b + 1
\]

\[
\text{elseif } a > b \text{ then}
\]

\[
\text{a} := a + 1
\]

\[
\text{else}
\]

\[
\text{b} := b + 1
\]

\[
\text{end}
\]

\[
\text{Conditional} \triangleq \{ \text{If} \text{ Then_part_list} \text{ ElseIf} \text{ end}
\]

\[
\text{Then_part_list} \triangleq \{ \text{Then_part} \text{ ElseIf} \}^*
\]

\[
\text{Then_part} \triangleq \{ \text{Boolean_expression} \text{ Then} \text{ Compound} \}
\]

\[
\text{Else_part} \triangleq \text{else Compound}
\]
BNF for simple arithmetic expressions

Is this a recursive grammar? Yes (see Nested)
How would the same grammar in BNF-E look like? (see yellow box below)
Which of the following phrases are correct?

a
a + b
-a + b
7 / (3 * 12) - 7
3 * 7
(5 + a (7 * b))

Guidelines for Grammars

Keep productions short.
easier to read
better assessment of language size
Guidelines for Grammars

Treat lexical constructs like terminals

Identifiers

Constant values

Identifier \(\triangleq\) Letter (Letter \| Digit \| "_")*

Integer_constant \(\triangleq\) Digit*

Floating_point \(\triangleq\) [-] Digit* "," Digit*

Letter \(\triangleq\) "A" \| "B" \| ... \| "Z" \| "a" \| ... \| "z"

Digit \(\triangleq\) "0" \| "1" \| ... \| "9"

Guidelines for Grammars

Use unambiguous productions.
Applicable production can be found by looking at one lexical element at a time

Conditional \(\triangleq\) if Then_part_list [ Else_part ] end

Compound \(\triangleq\) { Instruction }*

Instruction \(\triangleq\) Conditional \| Loop \| Call \| ...

Writing a Parser

One feature per Production

Concatenation:
Sequence of feature calls for Nonterminals, checks for Terminals

Choice:
Conditional with Compound per alternative

Repetition:
Loop
Writing a Parser: EiffelParse

Automatic generation of abstract syntax tree for phrase

Based on BNF-E

One class per production

Classes inherit from predefined classes AGGREGATE,

CHOICE, REPETITION, TERMINAL

Feature production defines Production

Writing a Parser: Tools

Yooc

Translates BNF-E to EiffelParse classes

Yacc / Bison

Translates BNF to C parser

BNF similar syntax descriptions

Unix/Linux: Synopsis of commands

SYNOPSIS


config_file] [-M pathlist] [-P pager] [-S section_list]

[section] name ...


config_file] [-M pathlist] [-P pager] [-S section_list]

[section] name ...
Eiffel syntax


http://www.gobosoft.com/eiffel/syntax/

What we have seen

A way to describe syntax: BNF

Three variants: BNF, BNF-E, graphical

A glimpse into recursion

End of lecture 9