Lecture 9: Describing the Syntax
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

<table>
<thead>
<tr>
<th>Sequence: instruction instruction instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: from compound invariant expression variant expression until expression loop compound end</td>
</tr>
<tr>
<td>Conditional: if condition then compound expression elseif condition then compound else expression loop compound end</td>
</tr>
</tbody>
</table>

---

*Intro. to Programming, lecture 9: Describing the syntax*
Flowcharts

Intro. to Programming, lecture 9: Describing the syntax
One-entry, one-exit

(Compound)

(Loop)

(Conditional)
Control structures at the machine level

Unconditional branch:

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

Conditional branch, for example:

\[ \text{BEQ} \; loc\_a \; loc\_b \; label \]
The equivalent of if-then-else

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

BEQ \( \text{loc}_a \text{ loc}_b \) 111

101 \( \ldots \) Code for \( \text{Compound}_2 \) \( \ldots \)

BR 125

111 \( \ldots \) Code for \( \text{Compound}_1 \) \( \ldots \)

125 \( \ldots \) Code for continuation of program \( \ldots \)
In programming languages: the Goto

\begin{itemize}
  \item \textbf{test condition goto if\_part}
  \item \textit{Compound\_2}
  \item \textbf{goto continue}
\end{itemize}

\begin{itemize}
  \item \textbf{if\_part:} \textit{Compound\_1}
  \item \textbf{continue:} \textit{... Continuation of program ...}
\end{itemize}

\begin{tikzpicture}[node distance=2cm, on grid]
  \node (a) at (0,0) {$a = b$};
  \node (b1) [below left of=a] {Compound\_1};
  \node (b2) [below right of=a] {Compound\_2};

  \draw [->] (a) -- (b1) node [midway, above] {True};
  \draw [->] (a) -- (b2) node [midway, above] {False};
\end{tikzpicture}
“Goto considered harmful”

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

**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 12
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 blvelee taht I cluod aulacity uesdnatnrd waht I was rdgnieg. The Paomnnehal Pweor of the Hmuan Mnid Aoccdrnig to a rscheearch at Cmabrigde Unervtisy, is deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer be in the rghit pclae. The rset can be a taotl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Ptrety 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
Why describe the syntax formally?

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
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)
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
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

class PERSON
feature
  age: INTEGER
  -- Age
end

Intro. to Programming, lecture 9: Describing the syntax
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 \texttt{if}, \texttt{then}, \texttt{end} or symbols such as the semicolon \texttt{;} or the assignment \texttt{:=}
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:

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:

BNF: $A \mid B$

Meaning: either $A$ or $B$
BNF Elements: Repetition

Graphical representation:

\[
\{ A \}^* 
\]

Meaning: sequence of zero or more A
BNF Elements: Repetition, once or more

Graphical representation:

\[ \{ A \}^+ \]

Meaning: sequence of one or more A
BNF elements: Overview

Concatenation: $A \ B$

Optional: $[ \ A \ ]$

Choice: $A \ | \ B$

Repetition (zero or more): $\{ \ A \ \}^*$

Repetition (at least once): $\{ \ A \ \}^+$

Intro. to Programming, lecture 9: Describing the syntax
A simple example

Example phrases:

- .76
- -.76
- 1.56
- 12.845
- -1.34
- 13.0

Translate it to written form!
A simple example

written in BNF:

digit \triangleq 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

float_number \triangleq [ - ] \{ digit \}^* . \{ digit \}^+
Conditional:

```
if condition then instruction [ else instruction ] end
```

written in BNF:

```
Conditional Δ
```

Intro. to Programming, lecture 9: Describing the syntax 36
BNF: Conditional with `elseif`

Conditional $\triangleq$ if Then_part_list [ Else_part ] end

Then_part_list $\triangleq$ Then_part { elseif Then_part }*

Then_part $\triangleq$ Boolean_expression then Compound

Else_part $\triangleq$ else Compound
Different Grammar for Conditional

```
Conditional  ::=  If_part  Then_part  Else_list  end

If_part      ::=  if  Boolean_expression

Then_part    ::=  then  Compound

Else_list    ::=  {  Elsif_part  }*  [  else  Compound  ]

Elsif_part   ::=  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 $\triangleq I$ [ don’t ] Verb Names [ Quant ]
Names $\triangleq$ Name [{, Name}* and Name]
Name $\triangleq$ tomatoes | shoes | books | football
Verb $\triangleq$ like | hate
Quant $\triangleq$ a lot | a little
BNF-E

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

**Concatenation**
\[ A \triangleq B \, C \, [\, D \, ] \]

**Choice**
\[ A \triangleq B \, | \, C \, | \, D \]

**Repetition**
\[ A \triangleq \{ \, B \, \text{terminal} \, ... \, \}^* \]
(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)

\[
\text{Conditional} \triangleq \text{if } \text{Then\_part\_list} \ [ \text{Else\_part} ] \ \text{end}
\]

\[
\text{Then\_part\_list} \triangleq \text{Then\_part} \ \{ \ \text{elseif} \ \text{Then\_part} \ \}\ *
\]

\[
\text{Then\_part} \triangleq \text{Boolean\_expression} \ \text{then} \ \text{Compound}
\]

\[
\text{Else\_part} \triangleq \text{else} \ \text{Compound}
\]
BNF-E: Conditional

Conditional \( \triangleq \) \textbf{if} Then\_part\_list \[ Else\_part \] end

Then\_part\_list \( \triangleq \) \{ Then\_part elseif \ldots \}^+

Then\_part \( \triangleq \) Boolean\_expression \textbf{then} Compound

Else\_part \( \triangleq \textbf{else} \) 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:

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

\[
\text{Compound} \triangleq \{ \text{Instruction} \quad ; \quad \ldots \}^*
\]

\[
\text{Instruction} \triangleq \text{Conditional} \quad | \quad \text{Loop} \quad | \quad \text{Call} \quad | \quad \ldots
\]
Recursive grammars

Production name can be used in its own definition

Definition of \texttt{Then_part_list} with repetition:

\[
\text{Then\_part\_list} \triangleq \{ \text{Then\_part} \quad \text{elseif} \quad \ldots\}^*
\]

Recursive definition of \texttt{Then\_part\_list}:

\[
\text{Then\_part\_list} \triangleq \text{Then\_part} \quad \text{elseif} \quad \text{Then\_part\_list}
\]
if \( a = b \) then
  \( a := a - 1 \)
  \( b := b + 1 \)
elseif \( a > b \) then
  \( a := a + 1 \)
else
  \( b := b + 1 \)
end

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

Then_part_list \( \triangleq \) { Then_part elseif ... }+

Then_part \( \triangleq \) Boolean_expression then Compound

Else_part \( \triangleq \) else Compound
BNF for simple arithmetic expressions

Is this a recursive grammar?
How would the same grammar in BNF-E look like?
Which of the following phrases are correct?
a
a + b
-a + b
a * 7 + b
7 / (3 * 12) - 7
(3 * 7)
(5 + a ( 7 * b))

Expr ≜ Term { Add_op Term }*
Term ≜ Factor { Mult_op Factor}*
Factor ≜ Number | Variable | Nested
Nested ≜ ( Expr )
Add_op ≜ + | -
Mult_op ≜ * | /
BNF for simple arithmetic expressions
(Solution)

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

Expr $\triangleq \{\text{Term Add$_{\text{op}}$ ...}\}^+$
Term $\triangleq \{\text{Factor Mult$_{\text{op}}$ ...}\}^+$
Factor $\triangleq \text{Number} | \text{Variable} | \text{Nested}$
Nested $\triangleq (\text{Expr})$
Add$_{\text{op}}$ $\triangleq$ + | -
Mult$_{\text{op}}$ $\triangleq$ * | /
Guidelines for Grammars

Keep productions short.

easier to read
better assessment of language size

Conditional $\triangleq$

if Boolean_expression then Compound
{ elseif Boolean_expression then Compound }*
[ else Compound ] end
Guidelines for Grammars

Treat **lexical constructs** like terminals

**Identifiers**

**Constant values**

**Identifier** $\triangleq$ **Letter** ($**Letter** \mid **Digit** \mid "_"$)$^*$

**Integer_constant** $\triangleq$ **Digit**$^+$

**Floating_point** $\triangleq$ $[-]$ **Digit**$^*$ "." **Digit**$^+$

**Letter** $\triangleq$ "A" \mid "B" \mid ... \mid "Z" \mid "a" \mid ... \mid "z"

**Digit** $\triangleq$ "0" \mid "1" \mid ... \mid "9"
Guidelines for Grammars

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

\[\text{Conditional } \triangleq \text{ if Then_part_list } [\text{ Else_part } ] \text{ end}\]

\[\text{Compound } \triangleq \{\text{ Instruction }\}^*\]
\[\text{Instruction } \triangleq \text{ Conditional } | \text{ Loop } | \text{ Call } | \ldots\]
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
Unix/Linux: Synopsis of commands

SYNOPSIS

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