Introduction

- Formal specifications: use mathematical methods to specify software requirements
- A formal language: make precise statements about software and hardware
- A grammar
  - Present the language
  - Consists of a set of
    - terminal symbols (e.g. int)
    - non-terminal symbols (e.g. **TYPE**)
    - productions (form: **NON_TERMINAL** $\rightarrow$ "some BNF expressions"

BNF

- Short for "**Bachus-Naur Form**"
- A way to present grammars
- Rules for writing a BNF expression

$$
\begin{aligned}
\text{BNF} \quad &\rightarrow \quad \Gamma \\
&\quad\;\; | \; \text{BNF BNF} \\
&\quad\;\; | \; \text{BNF} \mid \text{BNF} \\
&\quad\;\; | \; \text{BNF}^{*} \\
&\quad\;\; | \; \text{BNF}^{+} \\
&\quad\;\; | \; \text{BNF ?} \\
&\quad\;\; | \; \text{(BNF)}
\end{aligned}
$$

where let $\Gamma = \{a, b\}$ be a set of symbols

- ab means first a then b
- a|b means either a or b
- a^* means many or no repetitions of a
- a^+ means one or more repetitons of a
- a? means one or no repetiton of a
- (ab) means grouping together 'first a then b'

Examples

Let $\Gamma = \{a, b\}$

- $(\mathrm{ab})^*$ = or **ab** or **abab** or **ababab** or ...
- $(\mathrm{a} \mid \mathrm{b})?$ = or **a** or **b**
- $(\mathrm{a} \mid \mathrm{b})^+$ = **a** or **b** or **aa** or **ab** or **ba** or **bb** or ...
- ab^* = **a** or **ab** or **abb** or **abbb** or ...
- ab^+ = **ab** or **abb** or **abbb** or ...
- $\mathrm{ab}?$ = **a** or **ab**
- $\mathrm{a}^*(\mathrm{ab})^+\mathrm{b}^*$ = **aab** or **abb** or **aababb** or ...
- $(\mathrm{a} \mid \mathrm{b})?(\mathrm{a}^+\mathrm{b})^* \neq$ **abb**

((+|-)?{0-9})?.{0-9}$^{+}$(E(+|-)?{0-9}$^{+}$)?
where {0-9}=(0|1|2|3|...|9)
denotes strings that represent floating point numbers

= **-1.23E+11** or
= **.3** or
= **+3.456E-5** or
= ...
≠ **-45.45**

Test questions

- Given alphabet Γ={a,b} and BNF grammar (a|b)?(bab|aba)$^+$a*b*, indicate using *true* or *false* if the following strings are in the language defined that grammar.

(a) bab

(b) babbab

(c) bababa

(d) aaa

(e) bbabb

(f) ababab

(g) abbabaaabb

- Over the alphabet Γ={a,b,c}, *give the BNF grammar* which defines the language which contains the set of all strings that must contain the substring **abacab**.

BESTT

- Short for "**Basic Extended Simple Type Theory**"
- Allow us to express properties of software systems
- A strictly typed language

Types

- Types in BESTT

$$
\begin{array}{lll}
\text{TYPE} & \rightarrow & \text{ATOMICTYPE} \\
 & & \mid \text{TYPEVAR} \\
 & & \mid \text{TYPE} \rightarrow \text{TYPE} \\
 & & \mid \text{TYPE} * \text{TYPE} \\
 & & \mid \text{TYPE set} \\
 & & \mid \text{TYPE seq} \\
 & & \mid (\text{TYPE})
\end{array}
$$

ATOMICTYPE → int | bool | char | string | unit

TYPEVAR → An identifier not used in any other place

- In ASCII we mark type variables using a ', e.g. 'a or
- we use the Greek letters α, β, γ, ...

- Examples
  - int $*$ int
  - int $\rightarrow$ int
  - int seq
  - bool set
  - ('a $*$ 'b) seq
  - ('a $\rightarrow$ 'b) $*$ 'a seq $\rightarrow$ 'b seq
  - ('a $\rightarrow$ 'b) $\rightarrow$ 'a seq $\rightarrow$ 'b seq

Test questions

1. Write out in words what (('a → 'b) ∗ ('a → 'b)) seq means.
2. Write the formal type for the function *select* which takes a tuple containing of function of type 'a to a type bool and a sequence of type 'a and return a sequence of type 'a.
3. Define currying. Give the formal type of the curried version of *select* defined in the previous question.
4. The type 'a → 'b → 'c has 2 interpretations. Using words, describe both.

Answers

1. The type of a sequence of pairs where both pair components are of type function from value of type 'a to value of type 'b.
2. (('a → bool) * 'a seq) → 'a seq
3. ('a → bool) → 'a seq → 'a seq
4. 'a → ('b → 'c): type of a function which takes a value of type 'a and returns a value of type function from value of type 'b to value of type 'c or
   ('a → 'b) → 'c: type of a function which takes a value of type function from a value of type 'a to a value of type 'b and returns a value of type 'c

Declarations

- TYPEDECL $\rightarrow$ type TYPE
- DECLARATION $\rightarrow$ val NAME : TYPE
  Declare that some symbol with the name NAME is of type TYPE
- Examples
  - val a: int
  - val f: int $*$ int $\rightarrow$ int
  - val s: int set
  - val x: (int $*$ char) seq
  - val fs: $(\alpha \rightarrow \beta)$ set

Signatures

- A set of declarations grouped together
- The mathematical equivalent of *interfaces* in software
- Structure

SIGNATURE → signature NAME =
sig
TYPEDECL*
DECLARATION*
end

- Example

  The following presents the signature of a stack over some type 'a

```
signature STACK =
    sig
    type 'a stack
    val empty       : 'a stack
    val is_empty    : 'a stack -> bool
    val pop         : 'a stack -> 'a stack
    val push        : 'a * 'a stack -> 'a stack
    val size        : 'a stack -> int
    val top         : 'a stack -> 'a
end
```

Expressions

- Constructed using predeclared identifiers, constants and function symbols.
- Each expression has a type. EXP_α denotes an expression of some type α.
- Expressions EXP_α are constructed as follows:

$$
\begin{aligned}
\mathrm{EXP} \quad = \quad & \mathrm{ID}_\alpha \text{ a declared identifier} \\
& \mid \text{a constant of type } \alpha \\
& \mid \mathrm{EXP}_{\alpha_1,\alpha_2,\ldots,\alpha_n \to \alpha}\ (\mathrm{EXP}_{\alpha_1}, \mathrm{EXP}_{\alpha_2}, \ldots, \mathrm{EXP}_{\alpha_n})
\end{aligned}
$$

$\text{EXP}_{\alpha\to\beta} = \lambda\ \text{ID}_\alpha : \text{TYPE}_\alpha\ .\ \text{EXP}_\beta$

- $\lambda\ \text{ID}_\alpha : \text{TYPE}_\alpha\ .\ \text{EXP}_\beta$ is called **λ-abstraction**.
- fn x:int $\Rightarrow$ x $*$ x : an ASCII representation of λ x:int.x $*$ x

$$\begin{aligned} \text{EXP}_{\alpha set} \quad &= \quad \text{S ID}_\alpha : \text{TYPE}_\alpha\ .\ \text{EXP}_{\alpha\to bool} \\ &\quad\ \ |\ \text{empty} \end{aligned}$$

- **S** is the set constructor.
- An alternate notation: $\{\text{ID}_\alpha : \text{TYPE}_\alpha \mid \text{EXP}_{\alpha\to bool}\}$

$$\begin{aligned} \text{EXP}_{\alpha seq} \quad &= \quad \text{EXP}_{\alpha seq} \parallel \text{EXP}_\alpha \\ &\quad\ \ |\ \text{nil} \end{aligned}$$

- $\parallel$ is used to add an element to the end of a sequence.

Examples

- empty
- push(2,empty)
- top(push(2,empty))
- size(push(1,push(2,empty)))
- is_empty(empty)
- is_empty(size(empty)) "bad" example

Formulas and Predicates

- Formulas are expressions of type **bool**.
- Predicates are expressions of type **'a** $\rightarrow$ **bool**.
- Formulas are constructed as follows:

$$
\begin{array}{lcl}
\text{FORMULA} & \rightarrow & \text{FORMULA} \wedge \text{FORMULA} \\
& & \mid \text{FORMULA} \vee \text{FORMULA} \\
& & \mid \text{FORMULA} \rightarrow \text{FORMULA} \\
& & \mid \neg\ \text{FORMULA} \\
& & \mid \forall\ \text{ID:TYPE}\ .\ \text{FORMULA} \\
& & \mid \exists\ \text{ID:TYPE}\ .\ \text{FORMULA} \\
& & \mid \text{EXP}_{bool}
\end{array}
$$

- Examples
  - true
  - $\text{true} \wedge \neg\ (\text{false})$
  - $\text{true} \vee \text{false}$
  - $\forall$ n: int.even$(2 * \text{n})$ (where even: int $\rightarrow$ bool)
  - $(\neg\ \text{a} \wedge \text{b}) \rightarrow \text{c}$ (where a,b, and c are formulas)
  - $\neg$ (sum(x,y)) (where sum: int $*$ int $\rightarrow$ int) "bad" example

test question

Given the following signature and variable declarations

```
signature TOTO =
    sig
    val a1 : 'a
    val a2 : 'a -> bool
    val a3 : 'b -> ('a seq)
    val a4 : int -> 'a
end
val a5 : int
val a6 : bool
```

Are the following valid expressions?

(a) a2

(b) λ a5:int . a2(a5)

(c) a3(a2(a4(-3)))

(d) S x:int . a2(x)

Are the following valid formulas?

(a) a2

(b) $\forall$ a5:int . a2(a5)

(c) a6 $\wedge$ true

(d) $\exists$ x:int . a2(a4(x))

(e) $\neg$ (a5)

Assertions

- Associates an expression to an identifier
- Not an assignment as used in programming languages
- Not an expression, no type
- Assertion of types
  - type IDENTIFIER = TYPE
- Assertion of expressions
  - val NAME (: TYPE)? = EXP
  - fun NAME ((NAME(: TYPE)?)*) (: TYPE) = EXP

Examples (assertion of types)

- type int_pair = int * int
- type aa = int → int
- type int_seq = int seq
- type pair_seq = ('a * 'b) seq
- type mapper = ('a → 'b) * 'a seq → 'b seq
- type abc = 'a → 'b → 'c
- type ab = (('a → 'b) * ('a → 'b)) seq

Examples (assertion of expressions)

- val a = 3
- val F: int → int = fn x ⇒ x+1
- fun F(x:int):int = x+1
- val emptyset = empty
- val s: int set = S x:int . odd(x)
- val emptyseq = nil
- val x: int seq = y || 2 (assuming y:int seq)