Programming language theory

1 Concrete and abstract representations

Consider the string `2+3*4`. To a human reader it is obvious that this string is an arithmetic expression that evaluates to 14. In order to evaluate it the reader must analyze it and discover that it is a sum of two terms, that the first term is a number, and the second term is a product of two factors, each being a number. Then he computes the product and adds the terms. When we want a program to evaluate an arithmetic expression it must analyze the string and build a data structure representing the expression that is easy to evaluate. This data structure is called an *abstract representation* of the expression. You have learned to define suitable data types for such representations. If the numbers in the expression are integers the following Java classes could be used. One method, `int value( )` has been implemented. This function defines the meaning, the *semantics*, of the expressions.

```
interface Expr {
  int value();
}
class Add implements Expr {
  Expr expr1, expr2;
  Add(Expr expr1, Expr expr2) {
    this.expr1 = expr1;
    this.expr2 = expr2;
  }
  int value() {
    return expr1.value() + expr2.value();
  }
}
class Mul implements Expr {
  Expr expr1, expr2;
  int value() {
    return expr1.value() * expr2.value();
  }
}
class Int implements Expr {
  int value;
  int value() {
    return value;
  }
}
```

The constructors for `Mul` and `Int` have been omitted.

In mathematics such abstractions are used all the time, but we are seldom aware that we analyze mathematical formulae and build abstract representations in our heads in order to understand and manipulate them. In computer science and programming the use of these abstractions are more obvious.

1.1 Languages

In computer science, a *language* is a set of strings. A *string* is a finite sequence of symbols from an alphabet. An *alphabet* is a finite set of *symbols*. A symbol of an alphabet can, in principle, be anything. In our applications the symbols may be ordinary characters, or it may be key words like `if` and `while`, an identifier, a numeral, or operator symbols like `+` and `&&`. We often use Σ as a name for an alphabet, w as a name for a string, and L as a name for a language.

We say that a language is a *language on an alphabet* when all its string are made up with symbols from the alphabet.

> **Exempel.** The *binary alphabet* has just two symbols, { `0,1` }. The language of all English words is a language on the alphabet { `a,...,a,A,...,Z` }. It contains, e.g. the strings `a`, `an` and `alphabet`. $\Diamond$

If a language is finite it is, at least in principle, possible to describe it by enumerating the strings. So the language of all two-digit binary numerals is { `00,01,10,11` }. Most interesting languages are infinite, as the set of binary numerals. Sometimes you may be informal using three dots, { `0,1,2,...` }, but few programming languages provide such a notation.

A string may have 0 symbols. We call this string the *empty* string and use the name ϵ for it.

Since languages are sets the standard set operations like $\in, \cup$ and $\cap$ may be used. Sometimes it is convenient to consider an alphabet as language, i.e. the language of all single symbol strings on the alphabet.

We introduce some other operations on languages. If L_0 and L_1 are two languages then $L_0 \cdot L_1$ is the language obtained by concatenating each string in the first language with each string in the second language.

$$L_0 \cdot L_1 = \{w_0\, w_1 \mid w_0 \in L_0, w_1 \in L_1\}$$

Usually we will omit the concatenation operator and write $L_0\, L_1$. Further, if L is a language,

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^{n+1} &= L \cdot L^n, \qquad n \geq 0 \end{aligned}$$

The *Kleene star* or *Kleene closure* of a language L is denoted by $L^\star$. A string w belongs to $L^\star$ if $w \in L^n$ for some natural number n.

$$w \in L^\star \qquad \text{iff} \qquad \exists i. w \in L^i$$

If Σ is an alphabet we use $\Sigma^\star$ will be the language containing all strings on Σ.

1.2 Concrete grammars

Every string made up of digits and arithmetic operators is not an arithmetic expression; the string `**134+*+` could be a secret password but it is not an arithmetic expression. We need some rules to describe how to build expressions. One way is to use a *grammar*. The most common kind of grammar used to describe expressions and programs is a *context-free grammar*. One such grammar for arithmetic expressions with numbers, addition and multiplication symbols is

$$
\begin{array}{lll}
\mathit{expr} & \to & \mathit{number} \\
\mathit{expr} & \to & \mathit{expr} + \mathit{expr} \\
\mathit{expr} & \to & \mathit{expr} * \mathit{expr} \\
\mathit{number} & \to & \mathit{digit} \\
\mathit{number} & \to & \mathit{digit}\ \mathit{number} \\
\mathit{digit} & \to & 0 \\
\ldots & \to & \\
\mathit{digit} & \to & 9
\end{array}
$$

Each line is called a *production*. The strings in *italics* are called *nonterminal symbols*. When writing by hand we may use angle brackets to emphasize them, <expr>. The symbol `::=` is a *meta symbol* of the grammar used to separate the *target* nonterminal symbol to the left from a string of nonterminal and *terminal symbols* to the right, the *yield*. The terminal symbols belong to the alphabet of the language described by the grammar.

One of the nonterminal symbols is the *start symbol* of the grammar. If none is indicated you may assume that the target of the first rule is the start symbol.

A string of terminal symbols is *generated* by the grammar if there is a *derivation* starting with the start symbol and terminating with the string in question. In each step of the derivation one nonterminal target is replaced by a corresponding yield of a production. These are two different derivations of the string `2+3*4`.

$$
\begin{array}{l}
\mathit{expr} \to \mathit{expr} + \ \mathit{expr} \to \ \mathit{number} + \ \mathit{expr} \to \\
\qquad \mathit{digit} + \ \mathit{expr} \to \ 2 + \ \mathit{expr} \to \\
\qquad 2 + \ \mathit{expr} * \ \mathit{expr} \to \ 2 + \ \mathit{number} * \ \mathit{expr} \to \\
\qquad 2 + \ \mathit{digit} * \ \mathit{expr} \to \ 2 + 3 * \ \mathit{expr} \to \\
\qquad 2 + 3 * \ \mathit{expr} \to \ 2 + 3 * \ \mathit{number} \to \\
\qquad 2 + 3 * \ \mathit{digit} \to 2 + 3 * 4
\end{array}
$$

$$
\begin{array}{l}
\mathit{expr} \to \mathit{expr} * \ \mathit{expr} \to \ \mathit{expr} * \ \mathit{number} \to \\
\qquad \mathit{expr} * \ \mathit{digit} \to \ \mathit{expr} * 4 \to \\
\qquad \mathit{expr} + \ \mathit{expr} * 4 \to \ \mathit{digit} + \ \mathit{expr} * 4 \to \\
\qquad 2 + \ \mathit{expr} * 4 \to 2 + \ \mathit{number} * 4 \to \\
\qquad 2 + \ \mathit{digit} * 4 \to 2 + 3 * \ \mathit{number} \to \\
\qquad 2 + 3 * \ \mathit{digit} \to 2 + 3 * 4
\end{array}
$$

The grammar does not describe which abstract representation to build, but the first derivation makes it easy build something representing `2+(3*4)` while the other favors `(2+3)*4`. We say that the grammar is *ambiguous*. One way to resolve the conflict is to supply rules for the precedence and associativity of the operators. For arithmetic expressions we always give higher precedence to multiplication than addition. Thus the expression should be evaluated as $2 + (3 * 4)$. We specify that these operators associate to the left so that $2 * 3 * 4$ should be evaluated as $(2 * 3) * 4$ and not as $2 * (3 * 4)$. For addition and multiplication this does not affect the result but for subtraction and division it makes a difference; $(2 - 3) - 4 \neq 2 - (3 - 4)$.

There is a meta symbol for grammars, |, used to specify alternative yields for a single target. Also a * may be used to specify that a group of symbols in the target may be repeated zero or more times. If the group contains more than one symbol it must be surrounded by meta symbol parentheses. When some terminal symbol looks like an meta symbol we may surround terminal symbols with quotation marks. Using this we write

$$
\begin{array}{lll}
expr & \rightarrow & number \mid expr\ + expr \mid expr\ * expr \\
number & \rightarrow & digit\ digit^\star \\
digit & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

In a course on compilers you will learn how to construct an unambiguous grammar for arithmetic expressions:

$$
\begin{array}{lll}
expr & \rightarrow & term\ (+\ term)^\star \\
term & \rightarrow & factor\ (*\ factor)^\star \\
factor & \rightarrow & digit\ digit^\star \\
digit & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

1.3 Abstract grammars

An *abstract grammar* is class or type declarations that can be used to represent objects or values with an internal structure. An abstract grammar for expressions using Java classes was given above. With Haskell the corresponding definition is shorter. `Int` is a predefine type in Haskell so we use `Num` instead.

```
data Expr = Num Integer | Add Expr Expr | Mul Expr Expr
```

The semantics is defined by

```
value (Num n) = n
value (Add expr1 expr2) = value expr1 + value expr2
value (Mul expr1 expr2) = value expr1 * value expr2
```

In Haskell we may write `e::Expr` to indicate that `e` has type `Expr`. A mathematician would consider Expr to denote the set of all values that can be constructed with the constructors `Num`, `Add` and `Mul` and write $e \in \text{Expr}$.

1.4 Regular expressions

In unix shells, like *csh* and *bash*, you may use * and ? in file name patterns; `??*.java` will match all file names with at least two characters followed by the java extension. Some programs like *emacs* and the stream editor *sed* support a more sophisticated set of patterns called *regular expressions*. Those are extensions of the set of regular expressions that we introduce here.

Each regular expression denotes or describes a language. As an example $0 \mid 1 \cdot (0 \mid 1)^\star$ is a regular expression that denotes language of all binary numbers without unnecessary leading zeros. In this expression 0 and 1 are symbols of the alphabet of the language while $\mid$, $\cdot$ and * are operators that may occur in regular expressions and parentheses are used for grouping in the same way as in arithmetic expressions.

Of course there is a concrete grammar that describes how regular expressions are built.

$$
\begin{array}{lll}
expr & \rightarrow & expr\ \cdot\ expr \\
expr & \rightarrow & expr\ \mid\ expr \\
expr & \rightarrow & expr^* \\
expr & \rightarrow & (\ expr) \\
expr & \rightarrow & \emptyset \\
expr & \rightarrow & \epsilon \\
expr & \rightarrow & \sigma
\end{array}
$$

where σ is any symbol from the alphabet of the language.

This grammar is again ambiguous. We let * have the highest precedence followed by $\mid$ and the $\cdot$ as described by the following grammar.

$$\begin{array}{lll} expr & \rightarrow & term\,(\mid\ term)^\star \\ term & \rightarrow & factor(\cdot\ factor)^\star \\ factor & \rightarrow & symbol \mid \emptyset \mid \epsilon \mid factor^* \mid (term\ ) \end{array}$$

We prescribe that $\mid$ and $\cdot$ associates to the left but as for addition and multiplication in arithmetic expressions this will not matter when we consider the meaning of a regular expression in the next section.

According to this definition the following are regular expressions on the alphabet $\{a, b\}$:

$$\emptyset, \quad a, \quad ab, \quad ((a \mid b) \cdot a)a\,(ba)^\star$$

1.5 Semantics of regular expressions

Every regular expression on an alphabet Σ denotes a language on that alphabet. We define a function $\mathcal{L}$ that gives the meaning of each regular expression. The meaning function takes a regular expression as an argument and returns a language, i.e. a set of strings. The definition follows the structure of the expressions. We start with the base cases:

$$\begin{aligned} \mathcal{L}[\![\emptyset]\!] &= \emptyset \\ \mathcal{L}[\![\epsilon]\!] &= \{\epsilon\} \\ \mathcal{L}[\![\sigma]\!] &= \{\sigma\}, \qquad \sigma \in \Sigma \end{aligned}$$

We use square brackets around the function arguments instead of parentheses. We will explain why later. The first $\emptyset$ is a regular expression, while the second one is the standard mathematical symbol denoting the empty set. In the second line the first occurrence of ϵ is a regular expression while the second one it is the empty string and correspondingly in the third line.

If α and β are regular expressions then

$$\begin{aligned} \mathcal{L}[\![\alpha\,\beta]\!] &= \mathcal{L}[\![\alpha]\!]\mathcal{L}[\![\beta]\!] \\ \mathcal{L}[\![\alpha \mid \beta]\!] &= \mathcal{L}[\![\alpha]\!] \cup \mathcal{L}[\![\beta]\!] \\ \mathcal{L}[\![\alpha^*]\!] &= (\mathcal{L}[\![\alpha]\!])^* \end{aligned}$$

Some authors use $\cup$ instead of $\mid$ in regular expressions. It is obvious why. In this context it is an advantage to use different symbols in the concrete representation and the mathematical set expression .

Exempel. The regular expression $(0 \mid 1)^\star$ denotes the language of all binary strings. $\mathcal{L}[\![(0 \mid 1)^\star]\!] = (\mathcal{L}[\![(0 \mid 1)]\!])^\star = (\mathcal{L}[\![0]\!] \cup \mathcal{L}[\![1]\!])^\star = (\{0\} \cup \{1\})^\star = \{0, 1\}^\star = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$. $\Diamond$

Exempel. $(0 \mid (1(0 \mid 1)^\star))$ denotes the language of all binary numerals without extra initial zeros, $\{0, 1, 10, 11, 100, \dots\}$ $\Diamond$

1.6 Semantic functions

We have defined semantics for arithmetic expressions and regular expressions. In both cases we used a *semantic function*. This is one way to define semantics and it is called *denotational semantics*. The domain of the semantic function is called the *syntactic domain* and the range is called the *semantic domain*.

For regular expressions the semantic function takes a regular expression as an argument and returns a language. If Σ is an alphabet let $RE(\Sigma)$ be the set of all regular expressions over the Σ. $RE(\Sigma)$ is the syntactic domain of $\mathcal{L}$.

If S is a set then we define the *power set*, 2^S, of S to be the set of all subsets of S. This means that $2^{\Sigma^\star}$ denotes the set of all languages on Σ. So $2^{\Sigma^\star}$ is the semantic domain of $\mathcal{L}$ and

$$\mathcal{L} \in RE(\Sigma) \to 2^{\Sigma^\star}$$

For arithmetic expressions we defined semantic functions in Haskell and Java. The type of the Haskell function is `value :: Expr -> Integer`. In Java the "type" is given by the interface.

```
interface Expr {
  int value();
}
```

In a normal mathematics context we are always using some concrete grammar notation for abstract values and a concrete expression always denotes the meaning. We write $2 + 3 * 4 = 2 + 12 = 14$ since the meaning (or value) of $2 + 3 * 4$ is equal to the meaning of 14. We use $=$ do denote equality of meaning, not of concrete representation. If we want to express that the concrete representations are unequal we would write `"2+3*4"`$\neq$`"2+12"`.

Couldn't we use quotation marks when we the representation and not the meaning? Well, it would be misleading if we used in the definition of the semantic function. as in $\mathcal{L}(\text{"}\alpha \mid \beta\text{"}) = \mathcal{L}(\text{"}\alpha\text{"}) \cup \mathcal{L}(\text{"}\beta\text{"})$. In the left hand side $\alpha \mid \beta$ is a *pattern* for a concrete expression and in the right hand side α and β denote the matched regular expressions. To emphasize that a string denotes a syntactical value rather than its meaning we are using the square brackets, $\mathcal{L}[\![[\!]\alpha \mid \beta]$. We may consider $[\![]\!]$ as a function that takes a concrete representation and returns an abstract representation, possibly containing meta variables denoting subexpressions.