Surface Syntax

How a language appears to the user

We have so far talked about language paradigms (models) and languages from "inside" — that is, we have focused on the abstract form of syntactic expression ("abstract syntax") where the data structure that we have dealt with are trees (modelled as tree-shaped terms in a data type defined in our favourite programming language, which we are using as a meta-framework for defining concepts, syntactic and semantic).

However, most presentations of a programming language (or any language) usually are in the form of *sequences* of *characters* that form words, punctuation marks, numerals, white space and so on. In oral languages, we experience a sequence of recognisable sound units called phonemes. (In handwriting, or even when using different fonts, the same character can be rendered in different ways, but humans are able to, usually, recognise them quite clearly. Likewise, we are able to recognise phonemes out of the sound waves that are incident on our ears or hearing devices. These finer distinctions do not concern us here).

The written form of a language, as presented to the user, and as written/keyed in by the user , is its *surface syntax*. So far we have downplayed surface syntax, and highlighted instead the importance of abstract syntactic structure. However, a good language designer should pay attention to good surface syntax design.

Some good rules of thumb are to use keywords and symbols that the language user can easily relate with mathematical or logical operations, and to use words which convey the intuition of control constructs — "`let`", "`for`" and "`while`" are good examples. Some decades ago brevity was important, but now it is recognised that programming languages are as much for communicating ideas such as algorithmic solutions between people as from the programmer to the machine. Hence it may make sense to use "`mod`" (as in number theory) instead of the less intuitive symbol "`%`", when talking of the remainder of integer division. A particularly egregious but unfortunately rampant misuse of a symbol is "`!`" to mean negation.

From Sequences of Characters to Sequences of Tokens

We shall now study a part of the front end of a language translator (compiler or interpreter), which called a *tokeniser*. A tokeniser is fed a sequence of characters, for example those in a program file, or those keyed in by a programmer during an interactive session, and it outputs a sequence of *tokens,* flagging any erroneous input while doing so. A token is any lexically meaningful symbol that can appear in a program — for example, keywords such as "`let`", punctuation symbols such as "`:`" or "`->`", mathematical operators such as "`+`" and "`*`", parentheses such as "`(`", *identifiers* used as variables or type names, comment markers "`/*`" and even white space such as blanks, tabs and new line markers.

We mentioned that a tokeniser reports an error when it encounters a character that is inappropriate. Such errors are called *lexical errors*. Note that reporting errors is an extremely important part of a language processor, and this is not always performed well — the error messages may be confusing if a compiler writer is not careful. There are many instances where an error message is misplaced, or reported in a manner that makes the

programmer spend more time locating and fixing the problem than they reasonably should.

Writing a tokeniser can both be boring as well as tricky. So it is good to use a set of useful conceptual mechanisms to guide us. The best way to think about a tokeniser is to think of a Finite State Automaton (Machine) which can recognise exactly those sequences of characters that characterise a class of tokens, e.g., *alphanumeric identifiers*. Tokenisers can be thought of as consisting of a combination of such FSMs.

Tokenisers can actually do a bit more than just recognise a sequence of characters as a token — they can also perform some computation alongside. For example, when recognising a sequence of digits as an integer, the computation of the value of the integer can be piggy-backed onto the recognition process.

Let Σ be a set of characters, called an *alphabet*, with a, b, c, d being typical meta-variables ranging over Σ. For example, let $\Sigma = \{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'\}$. (We will typically use the name *Digit* to talk about this alphabet.) We can form the set of all non-empty numeric sequences of digits, called, *DigSeq*, characterised as follows:

If $d \in Digit$, then the singleton character string consisting of $d \in DigSeq$.
If $d \in Digit$, and $s \in DigSeq$ then $ds \in DigSeq$.

Note that d stands for a character in the alphabet *Digit* , s stands for a character string in *DigSeq* (strings are sequences of characters) and ds stands for the character string that begins with the character represented by d, followed by the rest of the characters in string s.

However, one may not want to include sequences which begin with the digit '0', other than just the singleton string "0". In this case, we could given a definition for the decimal representation of Natural Numerals (*NatNum*) as follows:

"0" $\in NatNum$.
If $d \in Digit - \{'0'\}$ and $s \in DigSeq$ or s is the empty sequence then $ds \in NatNum$.

Exercise: Draw a FSM to recognise exactly (the set of strings in) *NatNum*

There may be no particular reason to <u>prefix</u> a character to a string. Instead we may choose to affix a character at the <u>end</u> of a string. (Or maybe to concatenate two strings).
Let us explore defining *NatNum* by tacking on a character to the end of a string:

If $d \in Digit$, then the singleton character string consisting of $d \in NatNum$.
If $s \in NatNum - \{'0'\}$ and $d \in Digit$ then $sd \in NatNum$.

What is the advantage in looking at the set *NatNum* in this way?
For one, as the "machine" moves <u>forward</u>, consuming characters as it does, it can do two things: first, its state can characterise the sequence of characters seen so far (that is, the machine does not need to remember the characters precisely, but can instead use the state to summarily characterise the string already seen); second, it can perform a useful computation on the string seen so far and take this computation forward as it recognised a new character.
For example, when it reads the first character $d \in Digit$, it can set the value of the numeral seen so far to the digit d's value. Suppose it has seen the natural numeral string $s \in NatNum$ so far, and has computed the value of s to be n. Now if the next character it

reads is digit d, it can compute the value of the natural numeral string sd to be the value of d added to $10 \times n$.

Note that every fixed string, such as "let" can be recognised by a FSM. The accepting or final state of the FSM is reached exactly after the last letter of the particular string.

Let us characterise another useful set of strings that we find in programming languages.
Let $Alpha = \{'a', \ldots, 'z'\}$ and $Digits = \{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'\}$.
We can characterise the set of alphanumeric strings starting with a letter (used for *identifiers*) as

> If $a \in Alpha$, then the singleton character string consisting of $a \in Ident$.
> If $s \in Ident$ and $a \in Alpha \cup Digit$ then $sa \in Ident$.

Exercise: Draw a FSM to recognise exactly (the set of strings in) $Ident$

Regular Expressions

As we said earlier, we can writing a tokeniser by coding up FSMs in our favourite programming language. However, the process is both boring and error-prone. Fortunately, many decades ago, some noted mathematicians came up with the notion of *regular languages* (sometimes called rational languages), using which some noted and very intelligent computer scientists in the 70s wrote a mini-compiler that took in a specification of a regular language as input, and produced as output the code of a tokeniser which would recognise (accept) exactly the strings that belonged to the regular language given as input.
This tool was called "lex", and is now the standard front-end for almost every piece of software where we need to find or process strings of characters. There are many variants of the tool, "flex" for instance, or a version of lex for every target language — we will use OCaml-lex in this course.) I should add that should you ever think of writing a tokeniser from scratch, don't ... characterise the strings using regular expressions, and use a lex-based tool. You'll make less errors, have better and standard error messages, and if you realise you need to change the input, you don't have to do much work to recode the tokeniser.

Regular expressions find use in tools other than lex. For example, most text editors and Unix shells support searching for strings that match a regular expression. The command "grep" stands for "Global search for Regular Expression and Print matching lines".

Abstract Syntax of Core Regular Expressions

So what is a regular expression? Here we shall, in the spirit of Occam's Razor, only present the core idea — and leave you to study the extended version and its surface syntax, which you will need when using lex.

Let Σ be an alphabet. Regular Expressions over Σ ($Reg(\Sigma)$) are defined as follows:

- The symbol $\varnothing$ (denoting the empty set) is in $Reg(\Sigma)$
- The symbol ϵ (denoting the empty string) is in $Reg(\Sigma)$
- For each letter $a \in \Sigma$, that symbol (denoting the single character string) is in $Reg(\Sigma)$

- If expressions r_1, r_2 are in $Reg(\Sigma)$, then the expression $r_1 \cdot r_2$ is in $Reg(\Sigma)$ (Sometimes, we will omit writing the little dot ($\cdot$) — as is common when writing multiplicative operations).
- If expressions r_1, r_2 are in $Reg(\Sigma)$, then the expression $r_1 + r_2$ is in $Reg(\Sigma)$
- If expression r is in $Reg(\Sigma)$, then the expression $r*$ is in $Reg(\Sigma)$

> Note: The first three forms are the elementary constants. Then we have sequential composition, sum (a form of choice) and star (a simple version of repetition).
> The star is called "Kleene star" or "Kleene closure".
> We will see later that some useful extensions can be made to this core language to make it easier to write. For example, we will allow optionals, complementation, and intersection. However, the expressive power does not increase.

Semantics of core Regular Expressions

The meaning of a regular expression is a set of strings. We will define this formally for the core regular expressions defined above. (The meanings of the forms in the extended language are easy to understand once you get a clear grasp of the meanings of core regular expressions).

First some notation for useful operations on sets of strings.
Let ϵ denote the empty string (that is the string with no characters in it).
Let S_1, S_2 be two sets of strings. Then let $S_1 \cdot S_2 = \{s_1 s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$ denote the set of strings which are formed by concatenating a string from S_1 with a string from S_2.
Let $(S_i)_{i \in \mathcal{I}}$ denote a family of sets of strings S_i indexed over some denumerable set $\mathcal{I}$.
Then let $\bigcup_{i \in \mathcal{I}} S_i = \{s \mid \exists_{i \in \mathcal{I}} : s \in S_i\}$.
Let S be a set of strings. Let us define S^n for $n \geq 0$ as follows:

$$S^0 = \{\epsilon\}$$
$$S^{1+k} = S \cdot S^k$$

Note that S^n does <u>not</u> mean that given strings in S are repeated n times, but rather that it contains strings which can be cut into n segments such that each segment belongs to S (each of the segments may differ from one another).

- Regular expression $\varnothing$ denotes the empty set $\{\}$ of strings (over alphabet Σ). $[\![\varnothing]\!] = \{\}$
- Regular expression ϵ denotes the singleton set $\{\epsilon\}$ containing only the empty string (over alphabet Σ). $[\![\epsilon]\!] = \{\epsilon\}$
- For each letter $a \in \Sigma$, the regular expression a denotes the singleton set containing the character a, namely $\{a\}$. $[\![a]\!] = \{a\}$
- If expressions r_1, r_2 denote the sets S_1, S_2 respectively , then the regular expression $r_1 \cdot r_2$ denotes the set $S_1 \cdot S_2$ $[\![r_1 \cdot r_2]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$
- If expressions r_1, r_2 denote the sets S_1, S_2 respectively , then the regular expression $r_1 + r_2$ denotes the set $S_1 \cup S_2$ $[\![r_1 + r_2]\!] = [\![r_1]\!] \cup [\![r_2]\!]$
- If expression r denotes the set S , then the regular expression $r*$ denotes the set $\bigcup_{n \geq 0} S^n$

$$[\![r*]\!] = \bigcup_{n \geq 0} [\![r]\!]^n$$

The semantics allow us to state (and prove) various properties about regular expressions:

Associativity: $r_1 \cdot (r_2 \cdot r_3) = (r_1 \cdot r_2) \cdot r_3$
Identities: $\epsilon \cdot r = r = r \cdot \epsilon$
Annihilator: $\emptyset \cdot r = \emptyset = r \cdot \emptyset$
(Regular expressions for a monoid wrt $\cdot$ with identity ϵ and annihilator $\emptyset$)

Associativity: $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$
Identities: $\emptyset + r = r = r + \emptyset$
Commutativity: $r_1 + r_2 = r_2 + r_1$
Idempotence: $r + r = r$
(Regular expressions for a commutative, idempotent monoid wrt $+$, with identity $\emptyset$)

Left Distributivity: $r_1 \cdot (r_2 + r_3) = (r_1 \cdot r_2) + (r_1 \cdot r_3)$
Right Distributivity: $(r_1 + r_2) \cdot r_3 = (r_1 \cdot r_3) + (r_2 \cdot r_3)$
($\cdot$ left and right distributes over $+$)

This is called an idempotent semiring algebraic structure.

Exercise: Design a regular expression for integer numerals. Draw the FSM which allows optional (+, -) signs on the integer numerals and compare it with the regular expression.
Exercise: Design a regular expression for integer operators.
Exercise: Design regular expressions for boolean constants and operations.
Exercise: Design a regular expression for identifiers which are alphanumeric but may have underscores and primes (apostrophes). Define variants that start with capital letters and with small letters.
Exercise: Design a regular expression that can be used to eat up all white space.

Exercise: Modifying the regular expression for integer numerals, design regular expressions for real numbers with decimal points (and no unnecessary trailing zeroes after the decimal point), followed by a letter E and a signed integer exponent. Draw the FSM s and compare it with the regular expression you have written.

Exercise: Learn the language of extended regular expressions, and for each form try to define the standard denotational meaning. Test it out by searching for strings that match a regular expression in a very large file.

Exercise: Learn to use the tool lex. Note that lex takes in multiple regular expressions, each of which you will associate with a token class, and also provide functions that can compute values for the tokens. Note also that lex attempts to find the longest match. For example, even if "`let`" is a keyword, "`let1`" will be recognised as an alphanumeric identifier.

We resume our account of the idempotent semirings by defining an algebraic notion of ordering.
Definition: In an idempotent semiring, we define $r \leq r'$ if $r + r' = r'$
Note that if regular expressions $r \leq r'$, then and if S, S' denote the sets of strings represented by r, r', then $S \subseteq S'$.

Exercise: Prove that the ordering $\leq$ on regular expressions is a partial order, i.e., it is reflexive, transitive and anti-symmetric.
*__Exercise__: Prove that in an idempotent semiring, $r \leq r'$ if and only if for some r'', $r + r'' = r'$

Exercise: Prove that in an idempotent semiring, both operations $\cdot$ and $+$ are monotone with respect to the ordering $\leq$.

Exercise: Prove that in an idempotent semiring, the operation $+$ acts as a *least upper bound*, and $\emptyset$ is the least element of the ordering $\leq$. That is, $r \leq r + r'$ and $r' \leq r + r'$, and for *any* r'' such that $r \leq r''$ and $r' \leq r''$, we have $r + r' \leq r''$.
[Hint: The first part uses associativity, commutativity and idempotence of $+$, the "leastness" part uses only associativity of $+$ and the definition of $\leq$.]

Kleene Algebra

A Kleene Algebra is an idempotent semiring to which we add a unary iteration operation. The $\_^*$ operator provides the notion of zero or more iterations. We formulate its axioms as two (in)equations and two implications between (in)equations. Note our ordering inequations are actually equations.

Fix1: $\epsilon + rr^* \leq r^*$
Fix2: $\epsilon + r^*r \leq r^*$

LeastFix1: If $b + ar \leq r$ then $a^*b \leq r$
LeastFix2: If $b + ra \leq r$ then $ba^* \leq r$

In the above, for better readability, we have omitted writing the $\cdot$ operator.

We will not get into more detail about these axioms of the $\_^*$ operator other than to say that the rules together tell is that a^*b and ba^* are least fixed point solutions to the linear inequalities $b + ar \leq r$ and $b + ra \leq r$ respectively — and in fact the unique least fixed point solutions among all solutions in the Kleene algebra.

Fix1 says that a^*b is a solution to $b + ar \leq r$: By *monotonicity*, $\epsilon + aa^* \leq a^*$ implies $(\epsilon + aa^*)b \leq a^*b$. By distributivity and associativity, $b + a(a^*b) \leq a^*b$.
Similarly Fix2 (namely, $\epsilon + r^*r \leq r^*$) implies that ba^* is a solution to $b + ra \leq r$.

LeastFix1 says that if there is any other solution r' to $b + ar \leq r$ then $a^*b \leq r'$.
LeastFix2 says that if there is any other solution r' to $b + ra \leq r$ then $ba^* \leq r'$.

*Exercise: Prove that $\epsilon \leq r^*$
*Exercise: Prove that $r \leq r^*$
*Exercise: Prove that $r^*r^* \leq r^*$