CS320 High Level Computer Languages

CS320 Lecture/Discussion 03 -- Handouts

Your Project

In your project you will be developing a LRM (Language Reference Manual) for a programming language that you will be inventing. What makes a good LRM? Four things: Good comments, good examples, good syntax, and good semantics. Further they need to be intermingled: you should be able to see the reason for a feature, and example or two of the feature, and the rules (syntax and semantics) that apply to that feature close together. We will use the UML to also provide a visual summary of the semantics of the language.

So for each part of the language that you are defining you should have

1. Some comments
2. Some examples
3. Some syntax definitions using some kind of BNF (see over)
4. Some semantics....mainly English but sometimes a piece of the UML and/or math helps.

Like this:

Numbers are expressed using the familiar decimal notation. A string of decimal digits represents a number. The value of the number is calculated in the normal way.

1
512
8192

number ::= decimalDigit | decimalDigit number.
decimalDigit ::= "0".."9".

The meaning of a *decimalDigit* is called it's *value* and is a number:

$$value("0") = 0, value("1") = 1, \ldots value("9") = 9.$$

The meaning of a string of *n* digits $d = (d_n\ d_{n-1} \ldots d_3\ d_2\ d_1)$ is called it's *value* and

$$value(d) = \sum_{i=1}^{n} 10^{i-1} value(d_i)$$

For example, value(8192) = $8*10^3 + 1*10^2 + 9*10 + 2$.

Note. In the **final**, in your **project**, and in **class** I will often ask you to give me an *example* of something in a language. This means a piece of code that is in the language that shows that you know what it means. If I ask for

"an example of a C++ for loop"

then

```
for(i=1; i<=n; i++)
      v+=d[i-1];
```

is good but

```
for(....)...
```

is bad.

Syntax Notation in Projects.

Use XBNF as defined over the page OR you must define the notation you choose to use.

CS320 High Level Computer Languages

Cheat Sheet for the XBNF Metalanguage

This page summarizes a practical version of the Backus Naur Form (BNF) in the book. You will need it for the project, the final, and in class. It is called XBNF. This stands for eXtreme BNF because it is an eXtremely eXtended BNF. You can use it for more than syntax. I use it whenever I need to define anything.

. Example

BNF	<number>::=<digit>|<number> <digit>
EBNF	*number*::= *digit* {*digit*}.
XBNF	number::= N(digit).

More examples are on the WWW:

http://www.csci.csusb.edu/dick/samples/algol60.syntax.html
http://www.csci.csusb.edu/dick/samples/
http://www.csci.csusb.edu/dick/cs320/index.html#BNF

. Meta Symbols

The "::=" means "is defined to be". "|" separates alternatives. *Defined terms* (= *BNF non-terminal symbols*) have no <_>. *Terminal symbols* are written as C/C++/Java strings using double quotation marks. Parentheses "()" are used as they are in algebra. XBNF uses "#(_)" for `any number of, including zero`, "O(_)" for optional items, and "N(_)" for "one or more of". Three dots("...") indicate that something is defined somewhere else.

. Predefined XBNF Lexemes

You can use any of the following terms in defining syntax:
char::=`any ASCII character`. digit::= "0".."9". capital_letter::="A.."Z". letter::=capital_letter | "a.."z". underscore="_". sign::= "+" | "-". comma::=",", semicolon::=";", left_bracket::="[", right_bracket::="]". quotes::="\"". space::=" ". non_quote::=char ~ quotes. l_paren::="(", r_paren::=")",...

. Generalized Definitions

XBNF lets you define terms using any kind of mathematical expression. This lets it be used for semantics, glossaries, and dictionaries as well as syntax.

term ::=*expression*.
term ::=`*informal description*`.
For *parameters* , *term* ::=*expression*.
term ::*type* =*expression*.
term ::*type*, *properties_of_term.*
For *parameters* , *term* ::*type* =*expression*.

. Predefined XBNF Operations

The expressions in XBNF definitions are constructed using mathematical operators and functions.

Algebra + - * / ^ =(`equal to`) < > <= >= <>(`not equal to`) ...
Logic/Boolean and or not iff if...then..., for all...(...), for some ....(....).
Set Theory A | B::=union, A B::=concatenation, #A ::=`zero or more A's concatenated`,
A & B::=intersection, A ~ B::= A but not B, complement, @A::=`Power Set`,
A><B ::=`Cartesian product (set of pairs)`, %A ::= `Set of parenthesized Lists`,
A->B::=`The set of functions or mappings that given an A return a B`.
{ x : A || P(x) }::=`The set of all x in A that make P(x) true`.
For elements i, j, i..j ::= {k || i<=k<=j }, the set of k such that i <= k <= j,
(i..)::={k || k>=i}, (..j)::={k || k<=j}.

. Predefined sets

Boolean::={true, false}, Bit::={0,1}, Byte::=0..255, Natural::={1,2,3,...},
Unsigned::={0,1,2,3,...}, Integer::={..., -2,-1,0,1,2,...}
Rational::=`Ratios of an Integer and a Natural`, Real::=`See mathematical texts`,
Float(s) ::=`Rational with a power of two as a denominator and *s* significant bits`.

CS320 High Level Computer Languages

HTML::= "*HyperText Markup Language*".

This is a cheat sheet describing a subset of HTML suitable for completing CS320. For full details see http://www.csci.csusb.edu/dick/cs320/index.html#HTML.

HTML is a simple *markup language*. It lets you add *tags* like this: "<" *word* ">" to ASCII text to indicate how the text is to be interpreted. A typical *tag* is "<P>" that indicates a break between two paragraph. HTML is like RTF, XML, and other languages derived from SGML (Standardized General Mark up Language). The tags are often put paired around a piece of text:

<tag ... > text </tag >

An HTML file is interpreted by a *browser*. Simple HTML defines the desired structure and relative importance of the pieces of text. The person reading the page can use the browser to select a "look and feel" of the marked-up page. Later versions of HTML give the writer more control over what the reader sees. However, the user may be using a palm-top with a black-and-white non-graphic display. So stick to the simplest HTML, if you want everybody to see your work.

HTML uses "<", ">", *quotes*("), and *ampersand*(&) as special meta-characters. To represent these special characters in a page replace them by SGML *elements*: < > " &. There is a simple program ~dick/bin/ascii2html that will do this for you:

```
~dick/bin/ascii2html <file.txt >file.html
```

. Syntax

html_document::="<HTML>" *head body*, HTML documents have a *head* and a *body*.

For example

```
<HTML><head><title>Example of a simple HTML document</title></head>
<body> <h1>Hello, World!</h1>
</body>
```

The layout of the document does not usually matter -- this is indicated by tags, some of which are defined below.

head::="<head>" #(*title* | ...) "</head>". The head can include a *title*, styles, scripts, and meta-information.

title::="<title>"*ASCII_text* "</title>". The *title* is put in top of the window by browsers and used by search engines.

body::="<body>" #*piece* "</body>". The *body* is the part that is rendered and shown to the user.

The *body* of an HTML document is made of pieces an each piece has its own formatting. Typical pieces include headings, paragraph and line breaks, preformatted text, lists, tables, and so on.
piece::= *preformatted* | *break* | *elementary_piece* | *formatted_text* | *list* | *table* | *form*| ....

elementary_piece::=*text* | *image* | *anchor* | *applet* | ... .

preformatted::="<pre>" #*elementary_piece* "</pre>". Use end of lines and tabs, not
, <p>,... in pre-formatted text.

break::= "
" | "<p>" | "<hr>" |*heading* | ... . -- *br*=`line break`, *p*=`paragraph break`, *hr*=`horizontal rule`.

headings::=*heading*(1) | *heading*(2) | ...| *heading*(6). -- *heading*(1) is boldest and *heading*(6) least obtrusive.

```
<h1>Main Headline</h1>
<h6>Unimportant heading</h6>
```

For *n*: "1".."6", *heading*(*n*)::= "<h" *n* ">" *text* "</h" *n* ">" .

For *f*: *format*, *formatted_text*(*f*)::="<" *f* ">" #*piece* "</"*f*">".

format::= "blockquote"|"address" | "cite" | "em" | "strong" | ... . *em* =`emphasized`.

```
<address>5500, University Parkway, San Bernardino, CA 92407</address>
```

. Lists

list::="<dir>" *#item* "</dir>" | " " *#item* " "| " " *#item* " " | ... .
dir=`directory`, *ol*=`ordered list`, *ul*=`unordered list`.

```
<ol>
<li>Item 1
<li>Item 2
</ol>
```

item::= "<li>" #*piece*. -- Note. Items in a list can contain other lists.

```
<ol>
<li> Item 1
    <ul>
            <li>Item 1a
            <li>Item 1b
    </ul>
<li>Item 2
</ol>
```

. Graphics

image::="<img " "src=" *quotes file quotes* O("alt=" *quotes text quotes*) ... ">".
"img" indicates a graphic image to be inserted -- usually a "GIF" or "JPG" stored in the *src* file.

```
<img src="me.gif" alt="picture of cute baby">
```

. Hyperlinks

A piece of an HTML document can link or refer to another document, or a place in that document by using an *anchor*.
anchor::="" *#piece* "" | "" *#piece* "" .
href=`hypertext reference`.

. Uniform Resource Locators(URLs)

URL::= *protocol* ":" O(*location*) O(*label*)| ... . Some protocols don't allow labels. There is also a syntax for queries.

```
http://www.csci.csusb.edu/dick/cs320/index.html#HTML.
```

location ::= *relative_filename* | *absolute_name*. It is wise to always give a complete absolute name.
absolute_name::= O("//" *computer_name*) #("/" *directory*) O("/" O(*filename*)).
label::= "#" *identifier*.

protocol::= "http" | "ftp" | "telnet" | "mailto" | ...
http::="HypertText Transfer Protocol", *ftp*::="File Transfer Protocol".

. Applets

applet::="<applet" "code=" *class_name* ">" *#parameter* O(*text*) "</applet>", a little program executed by the browser.
*param*eter:= "<param" *name* "=" *value*">", data supplied to the applet.