CS 131 Programming Languages

September 29, 2009

Tail recursion: tail call to itself. Function f calls itself and returns whatever it returns. *Tail call*: f calls g and returns whatever g returns, no matter what. Tail calls are nice to optimize because compiler doesn't need to store return address on the stack.

Example: a recursive factorial function in OCaml.

```
(* not tail-recursive *)
def fact n =
       if n = 0 then 1
       else n * fact (n - 1);;

(* tail-recursive. returns n!*a *)
let facta n a =
       if n = 0 then a
       else facta (n-1) (n*a);;
(* worker function that calls facta *)
let fact n = facta n 1;;
```

Tail call above can be turned to GOTO statement by the compiler after it assigns new values to *n* and *a*. Although this structure looks like if-then-else, it is actually a while loop written in a different way.

Curried Functions

Named after Haskell Curry, a mathematician who came up with the notation. With the currying idea, you can specialize it without calling it. For example, instead of arcsin(x,y), an un-curried function, one can let arc3 = (arcsin 3) and then use arc3 as follows: arc3 4 = arcsin(3,4), etc.

Types

Redefining *list*: `type 'a list = [] | ('a) :: ('a list);;` *Discriminated union* – the method of storage of such object. Matching and using the list type: `match l with [] -> 0 | n::l -> n + foo l;;` Defining optional value: `type 'a option = None | Some of 'a;;` Using *option* type in a function: `let weird nopt = match nopt with None -> -1 | Some n -> 2  * n;;`

Spiglet

An intermediate language that represents partly compiled code; contains labels, multiplication, etc.

Syntax

Syntax – rules that tell whether a string belongs to certain language (is a sentence). An example from 1950's (M. Chomsky): "Colorless green ideas sleep furiously." This sentence is syntactically correct but its semantics are bogus. An opposite example: "Ireland has leprechauns galore." But *galore* is an adjective and therefore the syntax of this sentence is bogus, while semantics are good. Another example: "Time flies." Please go out to the field and time how fast flies are flying (not obvious meaning). Time passes by fast (expected meaning). Verb-noun or noun-verb? This sentence is syntactically and semantically ambiguous.

In programming, ambiguity is considered to be a bad thing – we'd like the computer to do exactly what we want.

Good Syntax

- What we're used to (semicolons)
- Unambiguous
- Concise
- Easy to read
- Easy to write (weaker goal)
    - APL is a classic example of easy-to-write, hard-to-read language.
- Simple

  - FORTH: no parentheses, no commas. Everything is done in reverse Polish notation. However, it's not what people are used to.
- Standardized
- Redundant – catches a lot of stupid mistakes (type errors, etc).

Definitions

Language is a set of sentences. *Sentence* is a finite string of tokens. *Token* is one of a finite sets of symbols (such as 0..255).

Example tokens for C++: `( ) * - -- / +  ++ if then else while return abc x-y3 297 2.31e-21`. Some tokens, when jammed together, are still distinct while others create a new token (such as `+-` versus `++`).

Reserved words can be problematic for upward compatibility reasons.

How can one prove that something is ambiguous (**midterm question**)? One may be able to find a valid sentence that could be interpreted in more than one way by the compiler. They were very careful with semicolons in PL1 (semicolon meant what follows it is a keyword) and that's why, even though they didn't have reserved words, they avoided parsing ambiguities.

Parsing a sentence using a parse tree: an English sentence can be split into its non-terminal symbols which can in turn be decomposed into (eventually) terminal symbols. Here's an example set of rules:

- `sentence -> np vp`
- `np -> n (noun phrase)`
- `np -> adj np (noun phrase)`
- `vp -> v (verb phrase)`
- `vp -> v do (verb phrase)`
- `do -> np (direct object)`

The homework asks to construct a random parse tree using a given grammar.

Example: Every piece of email in the world conforms to the IEEE standard RFC 2822. It specifies what constitutes a valid email using nothing more than *grammar*. Grammar for lines that begin with `"Message-ID: <abc$37.49@cs.ucla.edu>"` – a supposedly unique (in the world) ID of the message. The grammar that specifies this field looks like the following:

```
msg-id = "<" word *("." word) @ atom *("." atom) ">"
word = atom / quoted-string
atom = 1*<any CHAR except specials SPACE & CTLs>
```

However, this is not an entirely valid BNF statement. Tokens `*(".”` and `)` (*one or more of what's inside*) are not part of BNF, but they're part of EBNF (extended BNF). Token `/` (or) is not part of BNF either. Third line flakes out completely by using regular English to specify the grammar: CHAR means ASCII, specials are defined as `( ) / < > @ , ; : \ " . [ ]`

The goal of EBNF is to be easily (mechanically) translatable into BNF:

```
msg-id = "<" word dwlist "@" atom dalist ">"
dwlist =
dwlist = "." word dwlist
dalist =
dalist = "." word dalist
word = atom
word = quoted-string
```

Idea: translate RFC2822 into a grammar suitable for homework 1 and generate random message-ID headers.

Generating Random Grammar

The first rule (*sentence*) doesn't require a random generator since there's only one choice. The second rule (*np* or *vp*) requires a random value.

The random generator *seed* is walking through the tree in pre-order form (right-to-left). Each steps requires tracking where it came from and where it went to. The argument passing pattern for the seed will be different, more complex than the result passing pattern.