Com S 541 — Programming Languages 1 March 17, 1997

Grammar for the elp Interpreter for λProlog

The following grammar for λProlog is adapted from the files `elp.grm` and `elp.lex` in release 0.1x of elp, obtained from Amy Felty. It uses the notation of [Wat91, Chapter 2].

This grammar is ambiguous, and disambiguations are indicated by comments in the grammar following a percent sign (%).

1 Macrosyntax

Files follow the grammar for *File* below. The interpreter evaluates *Term*s from the grammar below.

File ::= *Module* | *Module* *File*

Module ::= *Module-Decl* *Module-Entry-List*
Module-Decl ::= `module` *lcid* `.`
Module-Entry-List ::= *Module-Entry* | *Module-Entry* *Module-Entry-List*
Module-Entry ::= `import` *Import-List* `.`
 | *Sig-Entry* `.`
 | *Fixity-Decl* `.`
 | *Term* `.`

Import-List ::= *lcid* | *lcid* *Import-List*

Sig-Entry ::= `type` *lcid* *Type-Expr* | `kind` *lcid* *Kind-Expr*
Fixity-Decl ::= `infixl` *Fixity* | `infixr` *Fixity* | `infix` *Fixity* | `prefix` *Fixity* | `postfix` *Fixity*
Fixity ::= *id-list* *Int-Literal*

Term ::= *Atom*
 | *Term* *Atom* % *application: left associative*
 | *Term* `::` *Term* % *list cons: right associcitive*
 | *Term* *Integer-Relation* *Term*
 | *Term* `is` *Integer-Computation* % *integer computation*
 | *Term* `isc` *Char-Computation* % *char computation*
 | *Term* `iss` *String-Computation* % *string computation*
 | *Term* `=` *Term* % *unification of terms*
 | *Term* `,` *Term* % *and: right associative*
 | *Term* `;` *Term* % *or: right associative*
 | *Term* `=>` *Term* % *implies: left associative*
 | *Term* `:-` *Term* % *reverse implies: left associative*
 | `pi` *Term* % *universal quantification*
 | `sigma` *Term* % *existential quantification*
 | *Integer-Computation* % *just forms a term; to compute use* `is`
 | *Char-Computation* | *String-Computation* % *also just form terms; use* `isc` *or* `iss`

Integer-Relation ::= `=:=` | `=\=` % *equal, not equal*
| `<` | `>` | `=<` | `>=`
Integer-Computation ::= *Atom* | `size` *String-Computation* | `ascii` *Char-Computation*
| `~` *Integer-Computation* % *negation*
| *Integer-Computation Integer-Op Integer-Computation*
Integer-Op ::= `+` | `-` | `*` | `div` | `mod`

Char-Computation ::= *Atom*
| `chr` *Integer-Computation*
| `char_of` *String-Computation Integer-Computation*

String-Computation ::= *Atom*
| `str` *Char-Computation*
| `substring` *String-Computation Integer-Computation Integer-Computation*
| *String-Computation* `^` *String-Computation* % *string concatenation*

Atom ::= `(` *Term* `)` | `(` *Term* `)` `:` *Type-Expr*
| *Int-Literal* | *String-Literal*
| `!` % *cut*
| *Quoted-Id* | *id*
| *id* `\` *Term* % *lambda abstraction: right associative*

id ::= *lower-case-id* | *Upper-Case-Id*
| *lower-case-id* `:` *Type-Expr* | *Upper-Case-Id* `:` *Type-Expr*
lcid ::= *lower-case-id* | *Quoted-Id*
id-list ::= *lcid* | *lcid id-list*

Type-Expr ::= *Type-Expr* `->` *Type-Expr* | *Type-Appl*
Type-Appl ::= *Type-Atom* | *Type-Appl Type-Atom*
Type-Atom ::= `o` | `int` | `string` | `char` | *lcid* | *Upper-Case-Id* | `(` *Type-Expr* `)`

Kind-Expr ::= `type` | *Kind-Expr* `->` *Kind-Expr*

2 Microsyntax

The microsyntax tells how tokens are formed, and explicitly treats each character (i.e., the tokens of the grammar below are characters [Wat91]).

File ::= (*Token* | *Comment* | *Blank*)*
Token ::= *lower-case-id* | *Quoted-Id* | *Upper-Case-Id* | *Int-Literal* |*String-Literal* | *Keyword*
| `.` | `(` | `)` | `:-` | `,` | `;` | `=` | `::` | `->` | `\` | `:` | `=>`
Keyword ::= `module` | `import` | `type` | `kind` | `infixl` | `infixr` | `infix`
| `prefix` | `postfix` | `int` | `string` | `char`

Id-Chars ::= *lower-case-start* | *Upper-Case-Start* | *Digit* | `'`

lower-case-id ::= lower-case-start Id-Chars*
lower-case-start ::= lower-case-letter
| `!` | `&` | `$` | `+` | `/` | `<` | `=` | `>` | `?` | `@` | `~` | `#` | `*` | `‘` | `-` | `|`
lower-case-Letter ::= `a` | `b` | ... | `z`

Upper-Case-Id ::= Upper-Case-Start Id-Chars*
Upper-Case-Start ::= Upper-Case-Letter | `_`
Upper-Case-Letter ::= `A` | `B` | ... | `Z`

Quoted-Id ::= `'` Quote-Chars* `'`
Quote-Chars ::= `'` | `^` | Blank

Int-Literal ::= Digit Digit*
Digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

String-Literal ::= `"` (String-Char | `^ ^` | `^ "`)* `"`
String-Char ::= “any character except `^` or `"`”

Comment ::= Character ::= “any character except newline”

Blank ::= “a blank, tab, or newline character”
NewLineChar ::= “a newline character”

References

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice-Hall, New York, N.Y., 1991.