Charity Grammar

This document gives the left-recursionless BNF grammar for Charity, used by Chirp's "near-predictive" parser. A few productions have not been left-factored (for clarity), but could easily be.

Productions which cause the grammar to fail to be purely *LL(1)* are marked with †.

This grammar was written by Marc Schroeder, but embodies the syntactic structure of the Chirp prototype written by Tom Fukushima, under Dr. Robin Cockett.

Conventions obeyed in the grammar below are as follows:

- The start symbols are labeled *ITEM*.
- Other non-terminals are labeled *item*.
- Reserved words are labeled `item`.
- Symbolic tokens are labeled "`item`"
- The "end of file" token is labeled specially as **eof**.
- Other terminals (ie. ones with attributes—identifiers and commands) are labeled **`item`**.
- The empty string is denoted by ϵ.

FILE →
    elementList **eof** .

elementList →
    element elementList
  | ϵ .

ADHOC† →
    element
  | *generalCommand* "`.`" .

element →
    dataDef "`.`"
  | *functionDef* "`.`"
  | *expression* "`.`"
  | *cDef* "`.`"
  | *cExpression* "`.`"
  | *specialCommand* "`.`"
  | "`.`" .

dataDef† →
    inductiveDef
  | *coinductiveDef* .

inductiveDef →
    `data` **`identifier`** *finiteProd* "`->`" **`identifier`** "=" *constructorList* .

```
finiteProd →
        "(" identList ")"
      | ε .

identList →
        identifier identList′
      | ε .

identList′ →
        "," identifier identList′
      | ε .

constructorList →
        constructor constructorList′ .

constructorList′ →
        "|" constructor constructorList′
      | ε .

constructor →
        identifier ":" type "->" identifier .

type →
        type′
      | type′ "*" type′
      | type′ "+" type′ .

type′ →
        "(" type ")"
      | "1"
      | identifier
      | identifier "(" identList ")" .

coinductiveDef →
        data identifier "->" identifier finiteProd "=" destructorList .

destructorList →
        destructor destructorList′ .

destructorList′ →
        "|" destructor destructorList′
      | ε .

destructor →
        identifier ":" identifier "->" type .

functionDef →
```

`def` **`identifier`** *formalMacroList formalParamList* “=” *expression* .

formalMacroList $\rightarrow$
“{” *identList* “}”
| ϵ .

formalParamList $\rightarrow$
pattern
| ϵ .

pattern $\rightarrow$
“`(`” *pattern′* “`)`”
| *pElement* .

pattern′ $\rightarrow$
pElement
| *pElement* “`,`” *pElement*
| *pElement* “`,`” “`(`” *pattern′* “`)`”
| “`(`” *pattern′* “`)`” “`,`” *pElement*
| “`(`” *pattern′* “`)`” “`,`” “`(`” *pattern′* “`)`”
| ϵ .

pElement $\rightarrow$
`identifier`
| “_” .

expression $\rightarrow$
functionOrVarOrMap
| *foldOrCaseOrAbs*
| *unfoldOrRecordOrOther* .

closedExpression $\rightarrow$
“`(`” *closedExpression′* “`)`” .

closedExpression′ $\rightarrow$
expression possibleSecond
| ϵ .

possibleSecond $\rightarrow$
“`,`” *expression*
| ϵ .

functionOrVarOrMap† $\rightarrow$
function
| *variable*
| *map* .

function →
  identifier *actualMacroList actualParamList* .

actualMacroList →
  “{” *abstractionList* “}”
| ϵ .

abstractionList →
  abstraction abstractionList′
| ϵ .

abstractionList′ →
  “,” *abstraction abstractionList′*
| ϵ .

abstraction →
  pattern “=>” *expression* .

actualParamList →
  closedExpression
| ϵ .

variable →
  identifier .

map →
  identifier “{” *mapElementList* “}” *closedExpression* .

mapElementList →
  abstraction abstractionList′ .

foldOrCaseOrAbs →
  “{” *foldOrCaseOrAbs′* “}” *closedExpression* .

foldOrCaseOrAbs′† →
  “ | ” *fold*
| *case*
| *abstraction* .

fold →
  foldElement “ | ” *foldElementList* .

foldElementList →
  foldElement “ | ” *foldElementList*
| ϵ .

foldElement →

`identifier` “`:`” *pattern* “=>” *expression* .

case →
caseElement caseElementList .

caseElementList →
“`|`” *caseElement caseElementList*
| ϵ .

caseElement →
`identifier` *possiblePattern* “=>” *expression* .

possiblePattern →
pattern
| ϵ .

unfoldOrRecordOrOther →
“`(`” *unfoldOrRecordOrOther′* .

unfoldOrRecordOrOther′† →
unfold
| *record* “`)`”
| “`)`”
| *closedExpression′* “`)`” .

unfold →
“`|`” *pattern* “=>” *unfoldElementList* “`)`” *closedExpression* .

unfoldElementList →
unfoldElement “`|`” *unfoldElementList′* .

unfoldElementList′ →
unfoldElement “`|`” *unfoldElementList′*
| ϵ .

unfoldElement →
`identifier` “`:`” *expression* .

record →
recordElement recordElementList .

recordElementList →
“`,`” *recordElement recordElementList*
| ϵ .

recordElement →
`identifier` “`:`” *expression* .

```
cExpression →
        ceval combinatorList .

combinatorList →
        combinator combinatorList′ .

combinatorList′ →
        ";" combinator combinatorList′
      | ϵ .

combinator →
        combinatorName combParamList .

combinatorName →
        identifier
      | "!" .

combParamList →
        "{" combParams "}"
      | ϵ .

combParams →
        combinatorList combParams′ .

combParams′ →
        "," combinatorList combParams′
      | ϵ .

cDef →
        cdef identifier formalMacroList "=" combinatorList .

specialCommand →
        command .

generalCommand →
        command .
```