An Oz Subset

This document describes a subset of Oz [VH04] that is designed to be slightly larger than the declarative kernel language of chapter 2 (of [VH04]). The subset includes several syntactic sugars, but not so many that manipulation becomes unwieldy.

The purpose of this subset, and the programs in the following directory:
`http://www.eecs.ucf.edu/~leavens/COP4020/reducer`
is to show how to define various manipulations of Oz programs. In particular, we provide definitions of Free and Bound Variable identifiers and desugaring into the declarative kernel language. Also provided is a definitional interpreter. All of these programs are written in Oz (although using a larger portion of the language).

In what follows, I provide the lexical (Section 1) and context-free (Section 2 on page 3) grammars for what the tools use in lexical analysis and parsing. I also define the abstract syntax trees (Section 3 on page 3) that are used in other tools (Section 4 on page 7).

Notational Conventions

Throughout this document we use the following conventions for describing grammars. The square brackets '[' and ']' surround optional parts of a production. Thus

⟨A⟩ ::= `example [` ⟨B⟩ `]` ⟨C⟩

means the same thing as

⟨A⟩ ::= `example` ⟨C⟩
| `example` ⟨B⟩ ⟨C⟩

Furthermore, the notation "..." following an optional construct means zero or more repetitions of the enclosed sentential form. Thus

⟨A⟩ ::= `example [` ⟨B⟩ `] ...` ⟨C⟩

means the same thing as

⟨A⟩ ::= `example` ⟨Bs⟩ ⟨C⟩
⟨Bs⟩ ::= ⟨B⟩ ⟨Bs⟩
| ⟨Empty⟩
⟨Empty⟩ ::=

Single quotes (' and ') are used to indicate when notations such as '`[`' and '`|`' are to be taken literally as terminal symbols, as opposed to their default interpretation as meta-level grammatical symbols.

1 Microsyntax for An Oz Subset

The lexical analyzer in `OzLexer.oz`, which uses the code in `LexerTools.oz`, does lexical analysis for the Oz subset. The lexer recognizes all the keywords (really reserved words) of the entire Oz language, as well as all the special keyword tokens. However, it currently does not handle the Oz `\insert` directive.

The microsyntax recognized by the lexer is given in Figure 1 on the following page. This grammar which is to be understood lexically, that is, unlike a normal grammar, no spaces or comments may appear between the characters of a token. Thus, in particular, in a ⟨label⟩, there can be no space or comment before the left parenthesis.

The microsyntax and the lexer do not not consider `unit` to be a keyword, unlike the Oz documentation, but merely an ⟨Atom⟩.

⟨microsyntax⟩ ::= [⟨lexeme⟩] ...
⟨lexeme⟩ ::= ⟨space⟩ | ⟨comment⟩ | ⟨token⟩
⟨token⟩ ::= ⟨varId⟩ | ⟨keyword⟩ | ⟨literal⟩ | ⟨label⟩

⟨space⟩ ::= ⟨non-nl-space⟩ | ⟨end-of-line⟩
⟨non-nl-space⟩ ::= a blank, tab, or formfeed character
⟨end-of-line⟩ ::= ⟨newline⟩ | ⟨carriage-return⟩ | ⟨carriage-return⟩ ⟨newline⟩
⟨newline⟩ ::= a newline character
⟨carriage-return⟩ ::= a carriage-return character

⟨comment⟩ ::= `?` | `%` [⟨non-eol⟩] ... ⟨end-of-line⟩
⟨non-eol⟩ ::= any character except a newline or carriage return

⟨varId⟩ ::= ⟨CapitalLetter⟩ [⟨AlphaNumeric⟩] ... | ⟨underbar⟩ [⟨AlphaNumeric⟩] ...
⟨CapitalLetter⟩ ::= An ISO capital letter, including the letters A through Z
⟨AlphaNumeric⟩ ::= ⟨CapitalLetter⟩ | ⟨lowercaseletter⟩ | ⟨digit⟩ | ⟨underbar⟩
⟨lowercaseletter⟩ ::= An ISO lower case letter, including the letters a through z
⟨underbar⟩ ::= `_`

⟨keyword⟩ ::= `andthen` | `at` | `attr`
| `case` | `catch` | `choice` | `class` | `declare` | `define` | `div` | `do`
| `else` | `elseif` | `end` | `export` | `fail` | `finally` | `for`
| `from` | `fun` | `functor` | `if` | `in` | `lazy` | `local` | `lock`
| `meth` | `mod` | `of` | `orelse` | `proc` | `prop` | `raise`
| `self` | `skip` | `then` | `thread` | `try`
| `(` | `)` | `[` | `]` | `{` | `}`
| `'|'` | `#` | `::=` | `'...'` | `=` | `.` | `:=` | `^|` | `'[]'` | `$`
| `!` | `~` | `+` | `-` | `*` | `/` | `@` | `<-`
| `,` | `!!` | `<=` | `==` | `\=` | `<` | `=<` | `>`
| `>=` | `=:` | `:` | `<:` | `=<:` | `>:` | `>=:` | `::` | `:::`

⟨literal⟩ ::= ⟨Atom⟩ | ⟨Bool⟩ | ⟨Int⟩ | ⟨Float⟩ | ⟨String⟩
⟨Atom⟩ ::= ⟨lowercaseletter⟩ [⟨AlphaNumeric⟩] ...
| `'` ⟨non-quote-eol⟩ [⟨non-quote-eol⟩] ... `'`
⟨non-quote-eol⟩ ::= any character that is a ⟨non-eol⟩ character except a `'`
⟨Bool⟩ ::= `true` | `false`
⟨Int⟩ ::= [`~`] ⟨digit⟩ [⟨digit⟩] ...
⟨Float⟩ ::= ⟨Int⟩ `.` ⟨numeric⟩ ⟨exponent⟩
⟨numeric⟩ ::= [⟨digit⟩] ...
⟨exponent⟩ ::= ⟨exponentChar⟩ ⟨Int⟩
⟨exponentChar⟩ ::= `E` | `e`
⟨String⟩ ::= `"` [⟨non-dq-eol⟩] ... `"`
⟨non-dq-eol⟩ ::= any character that is a ⟨non-eol⟩ character except a `"`

⟨label⟩ ::= ⟨Atom⟩ `(` | ⟨Bool⟩ `(` | ⟨varId⟩ `(`

Figure 1: The lexical grammar recognized by the tools in this directory, which is a subset of the Oz lexical grammar. Note that an ⟨Atom⟩ that has the same sequence of characters as a ⟨keyword⟩ or a ⟨Bool⟩ will be recognized as that ⟨keyword⟩ or a ⟨Bool⟩, unless it is quoted.

2 Context Free Grammar for the Oz Subset

The parser in `MyOzParser.oz` recognizes the grammar in Figure 2 on the next page, which builds on the lexical grammar as usual.

In the figure, the names in the right column are not part of the grammar, but instead show what type of abstract syntax tree(s) record(s) is (are) made from the production(s) on the left. Compare these with the abstract syntax definitions in Section 3.

3 Abstract Syntax Trees for the Oz Subset

The grammar in Figure 3 on page 5, describes the abstract syntax trees produced by the parser and used by other tools. These data structures are all records with distinguishing labels. Thus the grammar in the figure represents the abstract syntax of our Oz subset. For example the following Oz statement:

```
A = B
```

is represented by (parsed into) the following abstract syntax tree:

```
unifyStmt(varId('A') varId('B'))
```

Note that the grammar uses Oz atoms to represent variable identifiers.

A more complex example of an abstract syntax tree is given in Figure 4 on page 6.

```
⟨Program⟩ ::= [ ⟨Sequence⟩ ] ⟨Queries1⟩                                  program
⟨Queries1⟩ ::= [ ⟨Query⟩ ] ...
⟨Query⟩ ::= declare ⟨Sequence⟩ in ⟨Sequence⟩                            declareInQuery
    | declare ⟨Sequence⟩                                                declareQuery
⟨Sequence⟩ ::= ⟨Statement⟩ [ ⟨Statement⟩ ] ...                           seqStmt
⟨Statement⟩ ::= skip                                                    skipStmt
    | local ⟨varId⟩ in ⟨InStatement⟩ end                                localStmt
    | ⟨varId⟩ = ⟨varId⟩                                                 unifyStmt
    | ⟨varId⟩ = ⟨varId⟩                                                 unifyStmt
    | if ⟨Expression⟩ then ⟨InStatement⟩
      else ⟨InStatement⟩ end                                            ifStmt
    | case ⟨Expression⟩ of ⟨Pattern⟩
      then ⟨InStatement⟩ else ⟨InStatement⟩ end                         caseStmt
    | { ⟨Expression⟩ [ ⟨Expression⟩ ] ... }                             applyStmt
    | fun { ⟨Formals⟩ } ⟨Expression⟩ end                                namedFunStmt
    | thread ⟨Statement⟩ end                                            threadStmt
⟨InStatement⟩ ::= ⟨Sequence⟩
    | ⟨Pattern⟩ = ⟨Expression⟩ in ⟨Sequence⟩                            inStmt
⟨Expression⟩ ::= ⟨RelationalExp⟩
⟨RelationalExp⟩ ::= ⟨VBarExp⟩ [ ⟨RelationalOperator⟩ ⟨VBarExp⟩ ]         applyExp
⟨RelationalOperator⟩ ::= == | \= | < | =< | > | >=
⟨VBarExp⟩ ::= ⟨PoundExp⟩ [ '|' ⟨VBarExp⟩ ]                              recordExp
⟨PoundExp⟩ ::= ⟨AdditiveExp⟩ [ # ⟨AdditiveExp⟩ ] ...                    recordExp
⟨AdditiveExp⟩ ::= ⟨MultiplicativeExp⟩ [ ⟨AdditiveOp⟩ ⟨MultiplicativeExp⟩ ] ...  applyExp
⟨AdditiveOp⟩ ::= + | -
⟨MultiplicativeExp⟩ ::= ⟨NegateExp⟩ [ ⟨MultiplicativeOp⟩ ⟨NegateExp⟩ ] ...  applyExp
⟨MultiplicativeOp⟩ ::= * | / | div | mod
⟨NegateExp⟩ ::= [ ~ ] ... ⟨DotExp⟩                                     applyExp
⟨DotExp⟩ ::= ⟨TightOpExp⟩ [ .  ⟨TightOpExp⟩ ] ...                       applyExp
⟨TightOpExp⟩ ::= [ !!  ]  ... ⟨PrimaryExp⟩                             applyExp
⟨PrimaryExp⟩ ::= ( ⟨Expression⟩ )
    | ⟨varId⟩ | ⟨atomExp⟩                                              varId, atomExp
    | ⟨boolExp⟩                                                         boolExp
    | ⟨intLit⟩ | ⟨floatLit⟩                                            intLit, floatLit
    | ⟨String⟩                                                          recordExp
    | ⟨label⟩ [ Field ] ... )                                           recordExp
    | proc { $ ⟨Formals⟩ } ⟨Sequence⟩ end                               procExp
    | if ⟨Expression⟩ then ⟨Expression⟩
      else ⟨Expression⟩ end                                             ifExp
    | case ⟨Expression⟩ of ⟨Pattern⟩
      then ⟨Expression⟩ else ⟨Expression⟩ end                           caseExp
    | { ⟨Expression⟩ [ ⟨Expression⟩ ] ... }                             applyExp
    | thread ⟨Expression⟩ end                                           threadExp
⟨Field⟩ ::= ⟨Feature⟩ :  ⟨Expression⟩                                   colonFld
    | ⟨Expression⟩                                                      posFld
⟨Feature⟩ ::= ⟨atomExp⟩ | ⟨intLit⟩ | ⟨boolExp⟩
⟨Pattern⟩ ::= ⟨VBarPat⟩
⟨VBarPat⟩ ::= ⟨PoundPat⟩ [ '|' ⟨VBarPat⟩ ]                              recordPat
⟨PoundPat⟩ ::= ⟨PrimitivePat⟩ [ # ⟨PrimitivePat⟩ ] ...                  recordPat
⟨PrimitivePat⟩ ::= ⟨varId⟩ | ⟨atomExp⟩                                 varIdPat, atomPat
    | ⟨boolExp⟩                                                         boolPat
    | ( ⟨Pattern⟩ )
    | ⟨label⟩ [ ⟨PatField⟩ ] ... )                                      recordPat
⟨PatField⟩ ::= ⟨Feature⟩ :  ⟨Pattern⟩                                   colonFld
    | ⟨Pattern⟩                                                         posFld
⟨Formals⟩ ::= [ ⟨Pattern⟩ ] ...
```


Figure 2: Concrete syntax recognized by the parser, which is a subset of Oz.

⟨Program⟩ ::= `program(`⟨List ⟨Query⟩⟩ ⟨Position⟩`)`
⟨Query⟩ ::= `seqQuery(`⟨Statement⟩ ⟨Position⟩`)`
| `declareInQuery(`⟨Statement⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `declareQuery(`⟨Statement⟩ ⟨Position⟩`)`

⟨Statement⟩ ::= `skipStmt(`⟨Position⟩`)`
| `seqStmt(`⟨List ⟨Statement⟩⟩ ⟨Position⟩`)`
| `localStmt(`⟨VarIdRef⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `unifyStmt(`⟨VarIdRef⟩ ⟨Expression⟩ ⟨Position⟩`)`
| `ifStmt(`⟨Expression⟩ ⟨Statement⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `caseStmt(`⟨Expression⟩ ⟨Pattern⟩ ⟨Statement⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `applyStmt(`⟨Expression⟩ ⟨List ⟨Expression⟩⟩ ⟨Position⟩`)`
| `namedFunStmt(`⟨Atom⟩ ⟨List ⟨Pattern⟩⟩ ⟨Expression⟩ ⟨Position⟩`)`
| `inStmt(`⟨Pattern⟩ ⟨Expression⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `threadStmt(`⟨Statement⟩ ⟨Position⟩`)`

⟨Expression⟩ ::= ⟨VarIdRef⟩
| `atomExp(`⟨Atom⟩ ⟨Position⟩`)` | `boolExp(`⟨Bool⟩ ⟨Position⟩`)`
| `intLit(`⟨Int⟩ ⟨Position⟩`)` | `floatLit(`⟨Float⟩ ⟨Position⟩`)`
| `recordExp(atomExp(`⟨Atom⟩ ⟨List ⟨Field⟩⟩ ⟨Position⟩`)`
| `procExp(`⟨List ⟨Pattern⟩⟩ ⟨Statement⟩ ⟨Position⟩`)`
| `ifExp(`⟨Expression⟩ ⟨Expression⟩ ⟨Expression⟩ ⟨Position⟩`)`
| `caseExp(`⟨Expression⟩ ⟨Pattern⟩ ⟨Expression⟩ ⟨Expression⟩ ⟨Position⟩`)`
| `applyExp(`⟨Expression⟩ ⟨List ⟨Expression⟩⟩ ⟨Position⟩`)`
| `threadExp(`⟨Expression⟩ ⟨Position⟩`)`

⟨VarIdRef⟩ ::= `varId(`⟨Atom⟩ ⟨Position⟩`)`

⟨Pattern⟩ ::= `varIdPat(`⟨Atom⟩ ⟨Position⟩`)`
| `atomPat(`⟨Atom⟩ ⟨Position⟩`)` | `boolPat(`⟨Bool⟩ ⟨Position⟩`)`
| `recordPat(atomExp(`⟨Atom⟩`)` ⟨List ⟨PatField⟩⟩ ⟨Position⟩`)`

⟨Field⟩ ::= `colonFld(`⟨Feature⟩ ⟨Expression⟩ ⟨Position⟩`)`
| `posFld(`⟨Expression⟩ ⟨Position⟩`)`

⟨PatField⟩ ::= `colonFld(`⟨Feature⟩ ⟨Pattern⟩ ⟨Position⟩`)`
| `posFld(`⟨Pattern⟩ ⟨Position⟩`)`

⟨Feature⟩ ::= `atomExp(`⟨Atom⟩ ⟨Position⟩`)` | `intLit(`⟨Int⟩ ⟨Position⟩`)`
| `boolExp(`⟨Bool⟩ ⟨Position⟩`)`

⟨Position⟩ ::= `pos:pos(`⟨FileName⟩ ⟨Line⟩ ⟨Col⟩`)`
| `pos:pos(`⟨FileName⟩ ⟨Line⟩ ⟨Col⟩ ⟨FileName⟩ ⟨Line⟩ ⟨Col⟩`)`
| ⟨Empty⟩

⟨Empty⟩ ::=
⟨FileName⟩ ::= ⟨Atom⟩
⟨Line⟩ ::= ⟨Int⟩
⟨Col⟩ ::= ⟨Int⟩

Figure 3: Abstract syntax for an Oz Subset, based on the textbook's section 2.6 [VH04]. The optional ⟨Position⟩ field is used for error messages, but is often omitted in examples made by hand. The type ⟨String⟩ is the type of character strings in Oz, ⟨Atom⟩ is the type of atoms in Oz, ⟨Int⟩ is the type of Integers in Oz, etc.

The following Oz statement:

```
fun {AddToEach A#B Ls}
   case Ls of
     (X#Y)|T then ({Plus A X}#{Plus B Y})|{AddToEach A#B T}
   else nil
   end
end
```

is represented by the following record structure (with all positions omitted):

```
namedFunStmt('AddToEach'
             [recordPat(atomExp('#')
                        [posFld(varIdPat('A')) posFld(varIdPat('B'))])
              varIdPat('Ls')]
   caseExp(varId('Ls')
           recordPat(atomExp('|')
                     [posFld(recordExp(atomExp('#')
                                       [posFld(varIdPat('X'))
                                        posFld(varIdPat('Y'))]))
                      posFld(varIdPat('T'))])
           recordExp(atomExp('|')
                     [posFld(recordExp(atomExp('#')
                                       [posFld(applyExp(varId('Plus')
                                                        [varId('A')
                                                         varId('X')]))
                                        posFld(applyExp(varId('Plus')
                                                        [varId('B')
                                                         varId('Y')]))]))
                      posFld(applyExp(varId('AddToEach')
                                      [recordExp(atomExp('#')
                                                 [posFld(varId('A'))
                                                  posFld(varId('B'))])
                                       varId('T')]))])
           atomExp(nil)))
```

Figure 4: Example showing how Oz is parsed into the record structures (abstract syntax trees) from Figure 3 on the preceding page.

4 The Main Functions

The reducer provides the following main functions to users.

1. The function

   ```
   FreeVarIdsStmt: <fun {$ <Statement>}: <Set <String>>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 3 on page 5 and returns a set of all the variable identifiers that occur free in its argument. Its implementation is in `FreeVarIds.oz`. There are tests in the file `FreeVarIdsTest.oz`.

   See the file `TestOutput.txt` for results of testing.

2. The function

   ```
   BoundVarIdsStmt: <fun {$ <Statement>}: <Set <String>>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 3 on page 5 and returns a set of all the variable identifiers that occur bound in its argument. Its implementation is in `BoundVarIds.oz`. There are tests in the file `BoundVarIdsTest.oz`.

   See the file `TestOutput.txt` for results of testing.

3. The function

   ```
   DesugarStmt: <fun {$ <Statement>}: <Statement>
   ```

   takes a ⟨Statement⟩ in the grammar of Figure 3 on page 5 and returns a ⟨Statement⟩ in the subset of that grammar that represents Oz statements in the kernel language of the textbook [VH04, Section 2.3].

   An implementation is in `Desugar.oz`. There are tests in the file `DesugarTest.oz`.

   See the file `TestOutput.txt` for results of testing.

4. The function

   ```
   Reduce1: <fun {$ <Config>}: <Pair String Config>>
   ```

   takes a non-terminal configuration and returns a new rule name and a configuration for the next step in the operational semantics of Oz, starting at the given configuration.

   The implementation is in `Reducer.oz`. There are tests in the file `ReducerTest.oz`.

   See `Configuration.oz` for details on configurations.

References

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.