The Language ABS

BNF Converter

September 30, 2014

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of ABS

Identifiers

Identifiers *Ident* are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '` reserved words excluded.

Literals

String literals *String* have the form `"`x`"`}, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals *Integer* are nonempty sequences of digits.

TypeIdent literals are recognized by the regular expression `‘`upper (letter | digit | ’_’ | ”’)*`‘`

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in ABS are the following:

`Fut`	`Int`	`Rat`	`Unit`
`assert`	`await`	`builtin`	`case`
`class`	`data`	`def`	`else`
`export`	`extends`	`fimport`	`from`
`get`	`if`	`implements`	`import`
`in`	`interface`	`let`	`local`
`module`	`new`	`null`	`return`
`skip`	`suspend`	`then`	`this`
`type`	`while`		

The symbols used in ABS are the following:

<	>	_	,
.	;	*	=
(	)	|	{
}	=>	?	&
||	&&	==	!=
<=	>=	+	-
/	%	~	[
]	!		

Comments

Single-line comments begin with //.Multiple-line comments are enclosed with /* and */.

The syntactic structure of ABS

Non-terminals are enclosed between < and >. The symbols -> (production), | (union) and **eps** (empty rule) belong to the BNF notation. All other symbols are terminals.

```
Type              ->  Unit
                   |  Int
                   |  Rat
                   |  Fut < Type >
                   |  _
                   |  QualType
                   |  QualType < [AnnType] >
[AnnType]         ->  eps
                   |  AnnType
                   |  AnnType , [AnnType]
AnnType           ->  [Ann] Type
QualType          ->  [QualTypeIdent]
[QualType]        ->  eps
                   |  QualType
                   |  QualType , [QualType]
QualTypeIdent     ->  TypeIdent
[QualTypeIdent]   ->  eps
                   |  QualTypeIdent
                   |  QualTypeIdent . [QualTypeIdent]
Program           ->  ModuleDecl
ModuleDecl        ->  module QualType ; [Export] [Import] [AnnDecl] MaybeBlock
Export            ->  export [AnyIdent]
                   |  export [AnyIdent] from QualType
                   |  export *
                   |  export * from QualType
[Export]          ->  eps
                   |  Export ; [Export]
ImportType        ->  fimport
                   |  import
Import            ->  ImportType [AnyIdent] from QualType
                   |  ImportType * from QualType
[Import]          ->  eps
                   |  Import ; [Import]
AnyIdent          ->  Ident
                   |  TypeIdent
[AnyIdent]        ->  eps
                   |  AnyIdent
                   |  AnyIdent , [AnyIdent]
AnnDecl           ->  [Ann] Decl
Decl              ->  type TypeIdent = Type ;
                   |  data TypeIdent = [ConstrIdent] ;
                   |  data TypeIdent < [TypeIdent] > = [ConstrIdent] ;
                   |  def Type Ident ( [Param] ) = FunBody ;
                   |  def Type Ident < [TypeIdent] > ( [Param] ) = FunBody ;
                   |  interface TypeIdent { [MethSig] }
                   |  interface TypeIdent extends [QualType] { [MethSig] }
                   |  class TypeIdent { [BodyDecl] MaybeBlock [BodyDecl] }
                   |  class TypeIdent ( [Param] ) { [BodyDecl] MaybeBlock [BodyDecl] }
                   |  class TypeIdent implements [QualType] { [BodyDecl] MaybeBlock [BodyDecl] }
                   |  class TypeIdent ( [Param] ) implements [QualType] { [BodyDecl] MaybeBlock [Bo
ConstrIdent       ->  TypeIdent
                   |  TypeIdent ( [ConstrType] )
ConstrType        ->  Type
                   |  Type Ident
[ConstrType]      ->  eps
```