The Language shl

BNF-converter

13 kwietnia 2015

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 shl

Identifiers

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

Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

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

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 shl are the following:

`Boolean`	`CYA`	`DO`
`DONE`	`ELSE`	`FI`
`FOR`	`False`	`IF`
`IN`	`Integer`	`PRINT`
`REF`	`RETURN`	`RETURNED`
`SOLUTION`	`String`	`THEN`
`True`		

The symbols used in shl are the following:

`;`	`=`	`(`
`)`	`,`	`==`
`!=`	`<`	`>`
`<=`	`>=`	`+`
`-`	`*`	`/`
`++`	`--`	

Comments

Single-line comments begin with `<==3`.
Multiple-line comments are enclosed with `<=` and `=3`.

The syntactic structure of shl

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Prog*⟩ ::= `SOLUTION` ⟨*Blk*⟩ `CYA`

⟨*Blk*⟩ ::= ⟨*ListDec*⟩ ⟨*ListStm*⟩

⟨*Stm*⟩ ::= `FOR` ⟨*Ident*⟩ `IN` ⟨*Exp*⟩ `DO` ⟨*Blk*⟩ `DONE`
| `IF` ⟨*Exp*⟩ `THEN` ⟨*Blk*⟩ `FI`
| `IF` ⟨*Exp*⟩ `THEN` ⟨*Blk*⟩ `ELSE` ⟨*Blk*⟩ `FI`
| `RETURN` ⟨*Exp*⟩ `;`
| `PRINT` ⟨*Exp*⟩ `;`
| ⟨*Exp*⟩ `;`
| ⟨*Ident*⟩ = ⟨*Exp*⟩ `;`

⟨*ListStm*⟩ ::= ϵ
| ⟨*Stm*⟩ ⟨*ListStm*⟩

⟨*Dec*⟩ ::= ⟨*Typ*⟩ ⟨*Ident*⟩ `;`
| ⟨*Typ*⟩ ⟨*Ident*⟩ = ⟨*Exp*⟩ `;`
| ⟨*Typ*⟩ ⟨*Ident*⟩ `(` ⟨*ListFArg*⟩ `)` `DO` ⟨*Blk*⟩ `RETURNED`

⟨*FArg*⟩ ::= ⟨*Typ*⟩ ⟨*Ident*⟩
| `REF` ⟨*Typ*⟩ ⟨*Ident*⟩

⟨*ListFArg*⟩ ::= ϵ
| ⟨*FArg*⟩
| ⟨*FArg*⟩ `,` ⟨*ListFArg*⟩

⟨*ListDec*⟩ ::= ϵ
| ⟨*Dec*⟩ ⟨*ListDec*⟩

⟨*Typ*⟩ ::= `Integer`
| `Boolean`
| `String`

⟨*Exp*⟩ ::= ⟨*Exp*⟩ == ⟨*Exp2*⟩
| ⟨*Exp*⟩ `!=` ⟨*Exp2*⟩
| ⟨*Exp1*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ < ⟨*Exp3*⟩
| ⟨*Exp2*⟩ > ⟨*Exp3*⟩
| ⟨*Exp2*⟩ <= ⟨*Exp3*⟩
| ⟨*Exp2*⟩ >= ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ + ⟨*Exp4*⟩
| ⟨*Exp3*⟩ − ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ `*` ⟨*Exp5*⟩
| ⟨*Exp4*⟩ `/` ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp5*⟩ ::= ⟨*Ident*⟩ ++
| ⟨*Ident*⟩ −−
| ⟨*Ident*⟩ `(` ⟨*ListIParam*⟩ `)`
| ⟨*Ident*⟩
| ⟨*Constraint*⟩
| ⟨*Exp6*⟩

⟨*Exp1*⟩ ::= ⟨*Exp2*⟩

⟨*Exp6*⟩ ::= `(` ⟨*Exp*⟩ `)`

⟨*Constraint*⟩ ::= ⟨*Integer*⟩
| ⟨*BoolT*⟩
| ⟨*String*⟩

⟨*BoolT*⟩ ::= `True`
| `False`

⟨*IParam*⟩ ::= ⟨*Exp*⟩

⟨*ListIParam*⟩ ::= ϵ
| ⟨*IParam*⟩
| ⟨*IParam*⟩ `,` ⟨*ListIParam*⟩

```
SOLUTION
    Integer increase (Integer a)
    DO
        a++;
        RETURN a
    RETURNED
    Integer increaseR (REF Integer a)
    DO
        a++;
    RETURNED
    Integer a = 41 - 42;
    Integer counter0 = 0;
    FOR i IN 3
    DO
    IF a
        a = increase(a);
    ELSE
        increaseR(counter0);
    FI
    DONE
    <= After evaluation counter0 equals to 1 =3
    PRINT counter0
CYA
```

Język składnią bardzo przypomina PASCALA.
Wszystkie konstrukcje powinny być zrozumiałe:

- wyrażenia na zmiennych są zapożyczone z C++
- typy danych nazywają się podobnie jak w innych językach
- bloki podobnie jak w bashu ujęte są w znaczniki DO DONE, z wyjątkiem funkcji, które konczą się znacznikiem RETURNED
- przyjmowanie wartości przez referencję oznaczaone jest przez REF poprzedzający nazwę typu
- cały program ujety jest w znaczniki SOLUTION I CYA
- mnogość wyrażeń (Exp1..6) spowodowana jest kolejnością ich ewaluacji