Original Grammar:

```
<program> ::= <fdecls> <declarations> <statement_seq>.
<fdecls> ::= <fdec>; | <fdecls> <fdec>; | ϵ
<fdec> ::= def <type> <fname> (<params>) <declarations> <statement_seq> fed
<params> ::= <type> <var> | <type> <var> , <params> | ϵ
<fname> ::= <id>
<declarations> ::= <decl>; | <declarations> <decl>; | ϵ
<decl> ::= <type> <varlist>
<type> ::= int | double
<varlist> ::= <var>, <varlist> | <var>
<statement_seq> ::= <statement> | <statement>; <statement_seq>
<statement> ::= <var> = <expr> | if <bexpr> then <statement_seq> fi | if <bexpr> then
<statement_seq> else <statement_seq> fi | while <bexpr> do <statement_seq> od |
print <expr> | return <expr> | ϵ
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <term> % <factor> | <factor>
<factor> ::= <var> | <number> | (<expr>) | <fname>(<exprseq>)
<exprseq> ::= <expr>, <exprseq> | <expr> | ϵ
<bexpr> ::= <bexpr> or <bterm> | <bterm>
<bterm> ::= <bterm> and <bfactor> | <bfactor>
<bfactor> ::= (<bexpr>) | not <bfactor> | (<expr> <comp> <expr>)
<comp> ::= < | > | == | <= | >= | <>
<var> ::= <id> | <id>[<expr>]
<letter> ::= [a-z]
<digit> ::= [0-9]
<id> ::= <letter> | <id><letter> | <id><digit>
<number> ::= <integer> | <double>
```

Grammar in LL(1) Form:

```
<program> ::= <fdecls> <declarations> <statement_seq>.
<fdecls> ::= <fdec>;  <fdecls’> | ϵ
<fdecls’> ::=  <fdecls> | ϵ
<fdec> ::= def <type> <fname> (<params>) <declarations> <statement_seq> fed
<params> ::= <type> <var> <params’> | ϵ
<params’> ::= , <params> | ϵ
<fname> ::= <id>
<declarations> ::= <decl>; <declarations’> | ϵ
<declarations’> ::= <declarations> | ϵ
<decl> ::= <type> <varlist>
<type> ::= int | double
```

<varlist> ::= <var><varlist’>
<varlist’> ::= , <varlist> | ϵ
<statement_seq> ::= <statement> <statement_seq’>
<statement_seq’> ::= ; <statement_seq> | ϵ
<statement> ::= <var> = <expr> | if <bexpr> then <statement_seq> <statement’> | while <bexpr> do <statement_seq> od | print <expr> | return <expr> | ϵ
<statement’> ::= fi | else <statement_seq> fi
<expr> ::= <term> <expr’’>
<expr’> ::= + <term> | - <term>
<expr’’> ::= <expr’> <expr’’> | ϵ
<term> ::= <factor> <term’’>
<term’> ::= * <factor> | / <factor> | % <factor>
<term’’> ::= <term’> <term’’> | ϵ
<factor> ::= <id> <factor’> | <number> | (<expr>)
<factor’> ::= <var’> | (<exprseq>)
<exprseq> ::= <expr> <exprseq’> | ϵ
<exprseq’> ::= , <exprseq> | ϵ
<bexpr> ::= <bterm> <bexpr’>
<bexpr’> ::= or <bterm> <bexpr’> | ϵ
<bterm> ::= <bfactor> <bterm’>
<bterm’> ::= and <bfactor> <bterm’> | ϵ
<bfactor> ::= (<bfactor’> | not <bfactor>
<bfactor’> ::= (<bfactor’> <bterm’> <bexpr’>) | not <bfactor> <bterm’> <bexpr’>) | <term> <expr’’> <comp> <expr>)
<comp> ::= < | > | == | <= | >= | <>
<var> ::= <id> <var’>
<var’> ::= [<expr>] | ϵ
<letter> ::= [a-z]
<digit> ::= [0-9]
<id> ::= <letter><id’’>
<id’> ::= <letter> | <digit>
<id’’> ::= <id’> <id’’> | ϵ
<number> ::= <integer> | <double>

First and Follow Sets:

SYMBOL	FIRST	FOLLOW
<program>	. ; [a-z] def double if int print return while	$
<fdecls>	def ϵ	. ; [a-z] double if int print

		return while
<fdecls’>	def ϵ	. ; [a-z] double if int print return while
<fdec>	def	;
<params>	double int ϵ	)
<params’>	, ϵ	)
<fname>	[a-z]	(
<declarations>	double int ϵ	. ; [a-z] fed if print return while
<declarations’>	double int ϵ	. ; [a-z] fed if print return while
<decl>	double int	;
<type>	double int	[a-z]
<varlist>	[a-z]	;
<varlist’>	, ϵ	;
<statement_seq>	; [a-z] if print return while ϵ	. else fed fi od
<statement_seq’>	; ϵ	. else fed fi od
<statement>	[a-z] if print return while ϵ	. ; else fed fi od
<statement’>	else fi	. ; else fed fi od
<expr>	([a-z] doubleNum integerNum	) , . ;] else fed fi od
<expr’>	+ -	) + , - . ; < <= <> == > >=] else fed fi od
<expr’’>	+ - ϵ	) , . ; < <= <> == > >=] else fed fi od
<term>	([a-z] doubleNum integerNum	) + , - . ; < <= <> == > >=] else fed fi od
<term’>	% * /	%) * + , - . / ; < <= <> == > >=] else fed fi od

<term’’>	% * / ϵ	) + , - . ; < <= <> == > >=] else fed fi od
<factor>	([a-z] doubleNum integerNum	%) * + , - . / ; < <= <> == > >=] else fed fi od
<factor’>	([ϵ	%) * + , - . / ; < <= <> == > >=] else fed fi od
<exprseq>	([a-z] doubleNum integerNum ϵ	)
<exprseq’>	, ϵ	)
<bexpr>	(not	do then
<bexpr’>	or ϵ	) do then
<bterm>	(not	) do or then
<bterm’>	and ϵ	) do or then
<bfactor>	(not	) and do or then
<bfactor’>	([a-z] <double> <integer> not	) and do or then
<comp>	< <= <> == > >=	([a-z] doubleNum integerNum
<var>	[a-z]	) , ; =
<var’>	[ϵ	%) * + , - . / ; < <= <> = == > >=] else fed fi od
<letter>	[a-z]	% ( ) * + , - . / ; < <= <> = == > >= [[0-9] [a-z]] else fed fi od
<digit>	[0-9]	% ( ) * + , - . / ; < <= <> = == > >= [[0-9] [a-z]] else fed fi od
<id>	[a-z]	% ( ) * + , - . / ; < <= <> = == > >= [ ] else fed fi od
<id’>	[0-9] [a-z]	% ( ) * + , - . / ; < <= <> = == > >= [[0-9] [a-z]] else fed fi od

<id''>	[0-9] [a-z] ϵ	% ( ) * + , - . / ; < <= <> = == > >= [ ] else fed fi od
<number>	<integer> <double>	%) * + , - . / ; < <= <> == > >=] else fed fi od