Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*: The commonest examples of such tokens are IDENTIFIER and tokens whose instances are literal constants of some type.

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*.

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*.

In fact we've already seen an example of the use of BNF to specify such a token:

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \ . \ \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \ \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \ \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by by Sethi to specify unsigned floating point literals in a simple language.

Example of How to Write a Recursive Descent Parsing Method, and How Such a Method Creates a Parse Tree

-

-

Example of How to Write a Recursive Descent Parsing Method, and How Such a Method Creates a Parse Tree

- We will consider how to write a parsing method

  varDecl()

  for the nonterminal defined by this EBNF rule:

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl> } ;
            | Scanner IDENTIFIER = new Scanner '(' System . in ')' ;
```

-

Example of How to Write a Recursive Descent Parsing Method, and How Such a Method Creates a Parse Tree

- We will consider how to write a parsing method

  varDecl()

  for the nonterminal defined by this EBNF rule:

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl> } ;
            | Scanner IDENTIFIER = new Scanner '(' System . in ')' ;
```

- We will also look at how the method reads a syntactically valid <varDecl> and outputs a sideways parse tree of its sequence of tokens.

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
           |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();

    if (getCurrentToken() == IDENT) nextToken();
    else throw new SourceFileErrorException("Scanner name expected");

    accept(BECOMES); accept(NEW); accept(SCANNER);
    accept(LPAREN); accept(SYSTEM); accept(DOT);
    accept(IN);  accept(RPAREN); accept(SEMICOLON);
  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();
  if (getCurrentToken() == INT) {









  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");
  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();
  if (getCurrentToken() == INT) {
    nextToken();









  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");
  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();
  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();





  }
  else if (getCurrentToken() == SCANNER) {





  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");
  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();
  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {



    }

  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");
  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();

    }

  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl()
    }

  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA != SEMICOLON) {  // ALTERNATIVE CODE
      nextToken(); accept(COMMA);
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA != SEMICOLON) {  // ALTERNATIVE CODE
      nextToken(); accept(COMMA);
      singleVarDecl();
    }
    accept(SEMICOLON); nextToken();                    // OPTIONAL CHANGE
  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {









  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();





  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();

    accept(IDENT);  // This is OK, but if currentToken != IDENT then the
                    // “Something’s wrong” error message is not very nice!




  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();

    if (getCurrentToken() == IDENT) nextToken(); // better than accept(IDENT);
    else throw new SourceFileErrorException("Scanner name expected");




  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;

private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();

    if (getCurrentToken() == IDENT) nextToken(); // better than accept(IDENT);
    else throw new SourceFileErrorException("Scanner name expected");

    accept(BECOMES); accept(NEW); accept(SCANNER);
    accept(LPAREN); accept(SYSTEM); accept(DOT);
    accept(IN);  accept(RPAREN); accept(SEMICOLON);
  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`
`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: z
  ... node has no more children
 ;
 ... node has no more children
```

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`
`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: z
  ... node has no more children
 ;
 ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken()==SCANNER) {
                ...
  }
  else
    throw new SourceFileErrorException
      ("\"int\" or \"Scanner\" needed");

  TJ.output.decTreeDepth();
}
```

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> **::= int** <singleVarDecl> **{ ,** <singleVarDecl>**}** ;

| **Scanner** IDENTIFIER = **new Scanner** '(' **System . in** ')' ;

```
private static void varDecl()
        throws SourceFileErrorException
{

}
```

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`

`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

<varDecl>

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);

}
```

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`
`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

<varDecl>

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`

`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

<varDecl>

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {





  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`

`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

<varDecl>

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();
  if (getCurrentToken() == INT) {





  }
  else if


  else


  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;

| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

<varDecl>

Reserved Word: int

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();




  }
  else if


  else


  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> {, <singleVarDecl>} ;
| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

<varDecl>
Reserved Word: int
<singleVarDecl>
IDENTIFIER: w
... node has no more children

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();




  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;

| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {


    }

  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;
| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

<varDecl>
Reserved Word: int
<singleVarDecl>
IDENTIFIER: w
... node has no more children
,

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();

    }

  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;
| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }

  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

... node has no more children

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;
| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }

  }
  else if


  else



  TJ.output.decTreeDepth();
}
```

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl> } ;
| **Scanner** IDENTIFIER = **new Scanner** '(' **System** . **in** ')' ;

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: z
  ... node has no more children

 ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }

  }
  else if


  else


  TJ.output.decTreeDepth();
}
```

Creation of Sideways Parse Tree of int w, x = 17, y[], z; **with root** <varDecl>

Based on <varDecl> ::= int <singleVarDecl> { , <singleVarDecl> } ;
| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: z
  ... node has no more children
 ;
 ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if


  else


  TJ.output.decTreeDepth();
}
```

Creation of Sideways Parse Tree of `int w, x = 17, y[], z;` **with root** `<varDecl>`

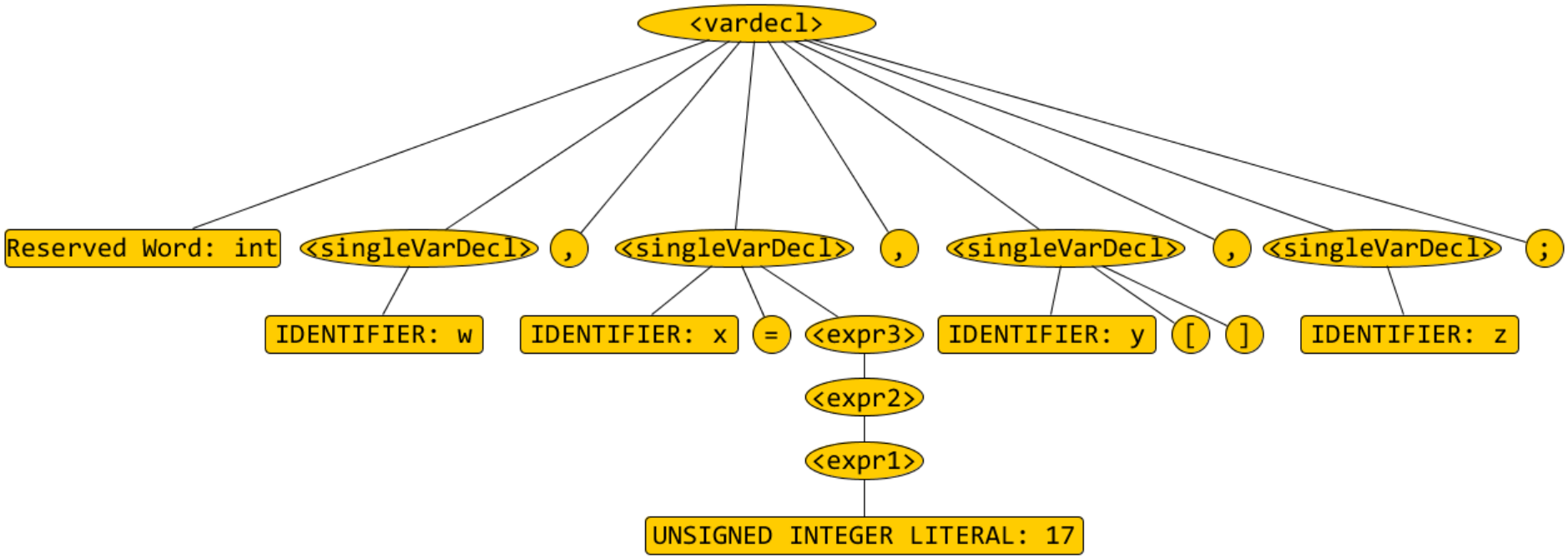
Based on `<varDecl> ::= int <singleVarDecl> { , <singleVarDecl>} ;`
`| Scanner IDENTIFIER = new Scanner '(' System . in ')' ;`

```
<varDecl>
 Reserved Word: int
 <singleVarDecl>
  IDENTIFIER: w
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
   <expr2>
    <expr1>
     UNSIGNED INTEGER LITERAL: 17
     ... node has no more children
    ... node has no more children
   ... node has no more children
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: y
  [
  ]
  ... node has no more children
 ,
 <singleVarDecl>
  IDENTIFIER: z
  ... node has no more children
 ;
 ... node has no more children
```

```
private static void varDecl()
        throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl);
  TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if


  else


  TJ.output.decTreeDepth();
}
```

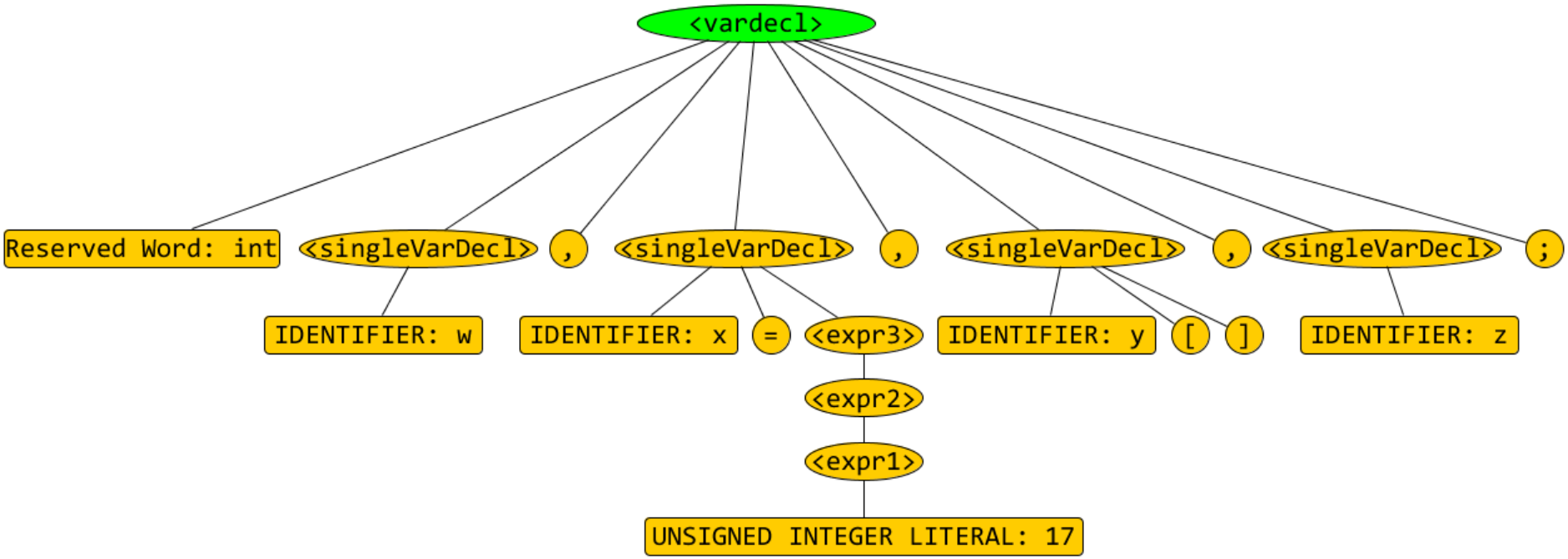
Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

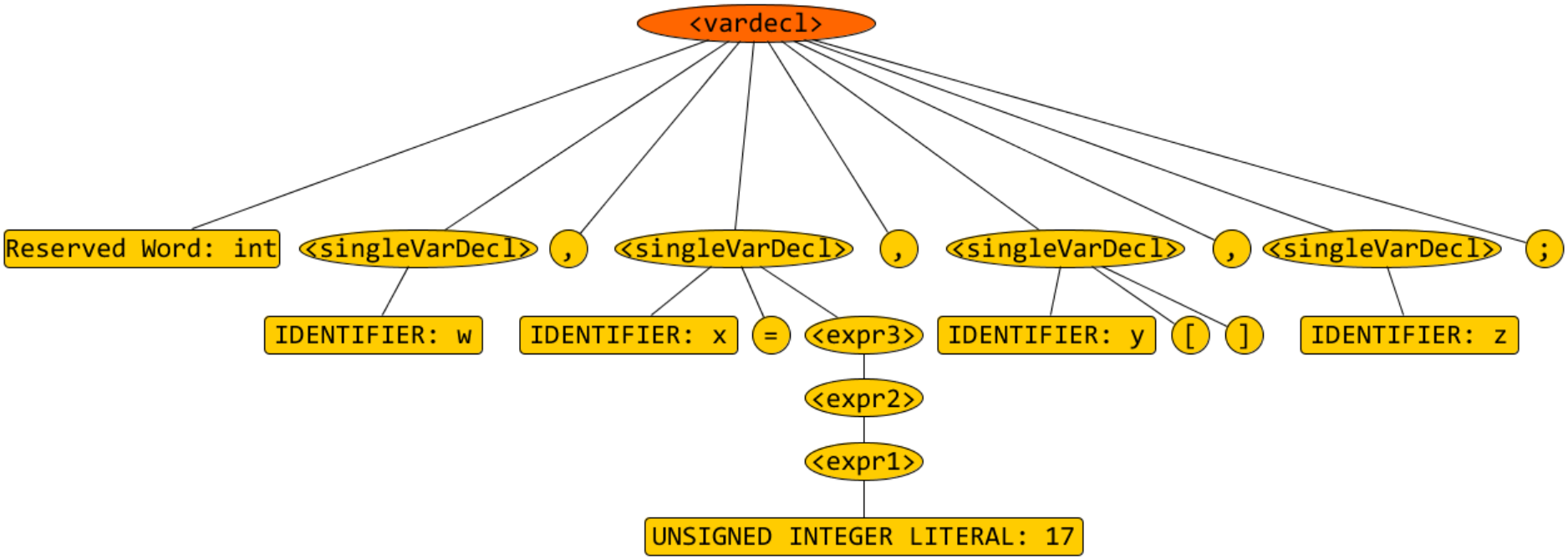
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `TJ.output.printSymbol(NTvarDecl);`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

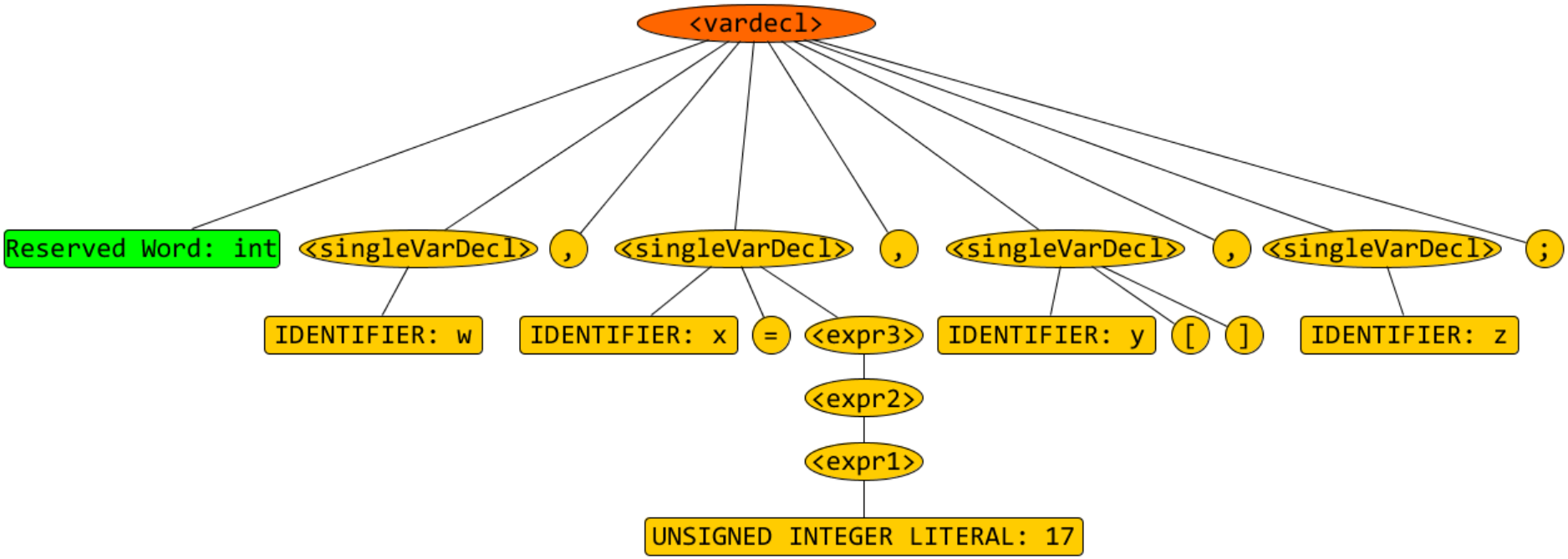
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

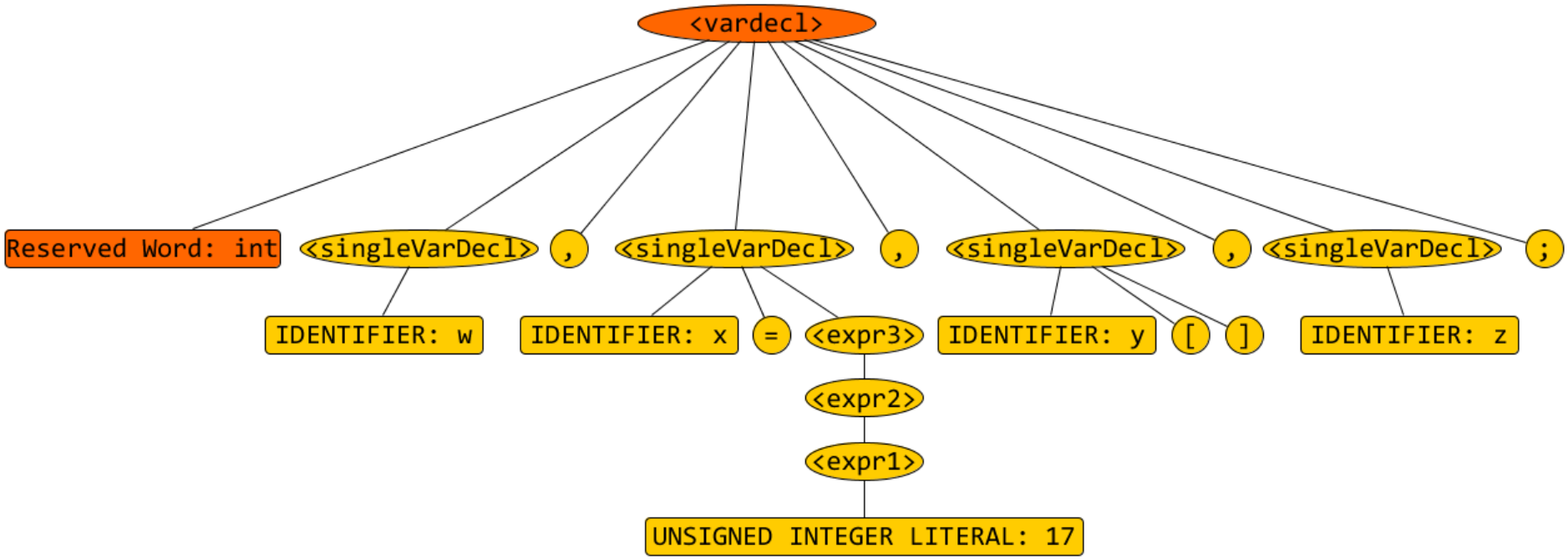
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `nextToken();`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

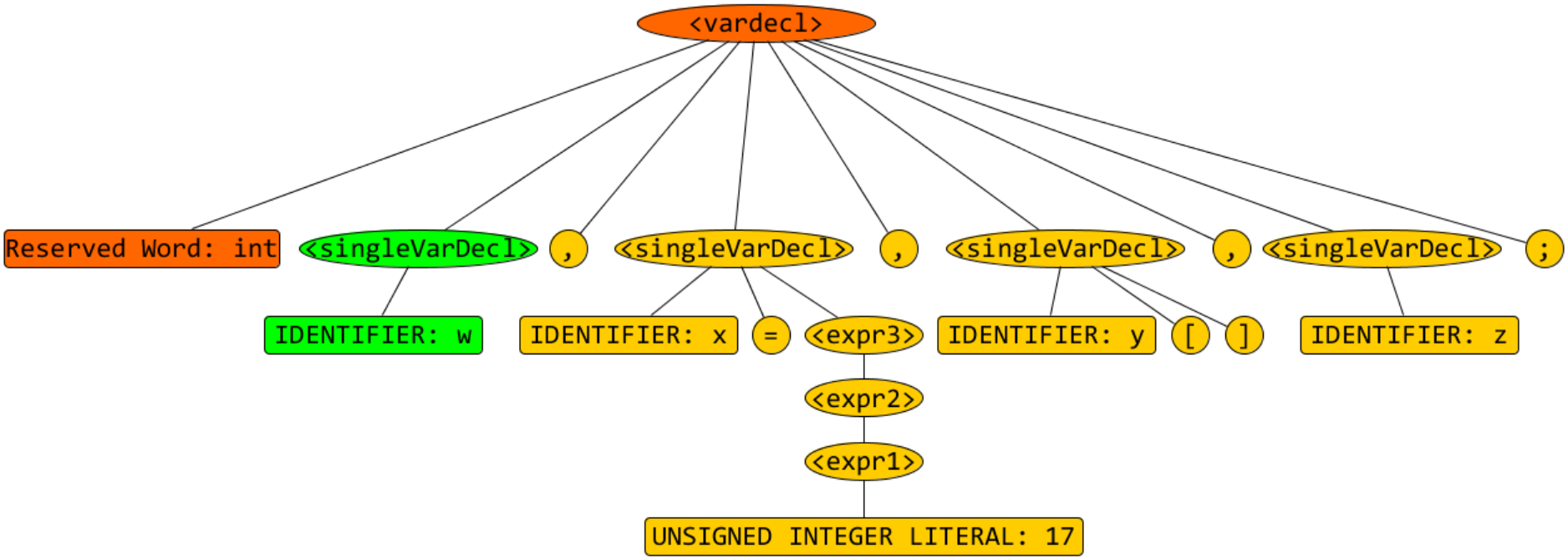
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

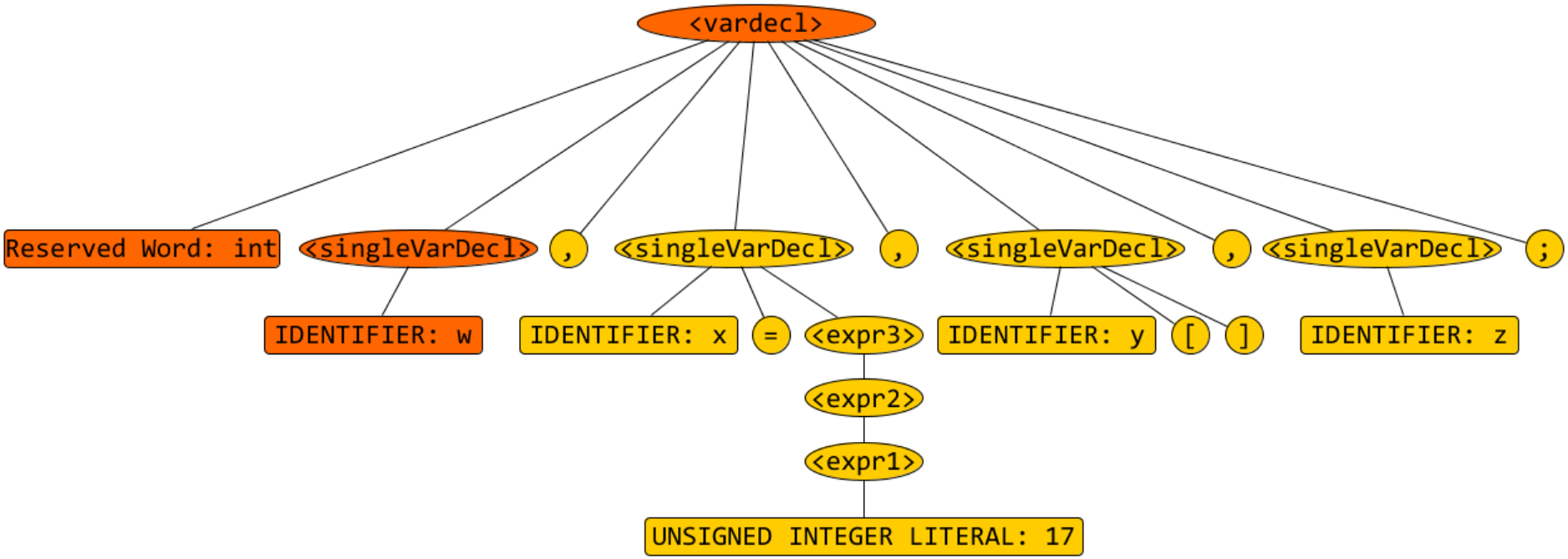
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `singleVarDecl();`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

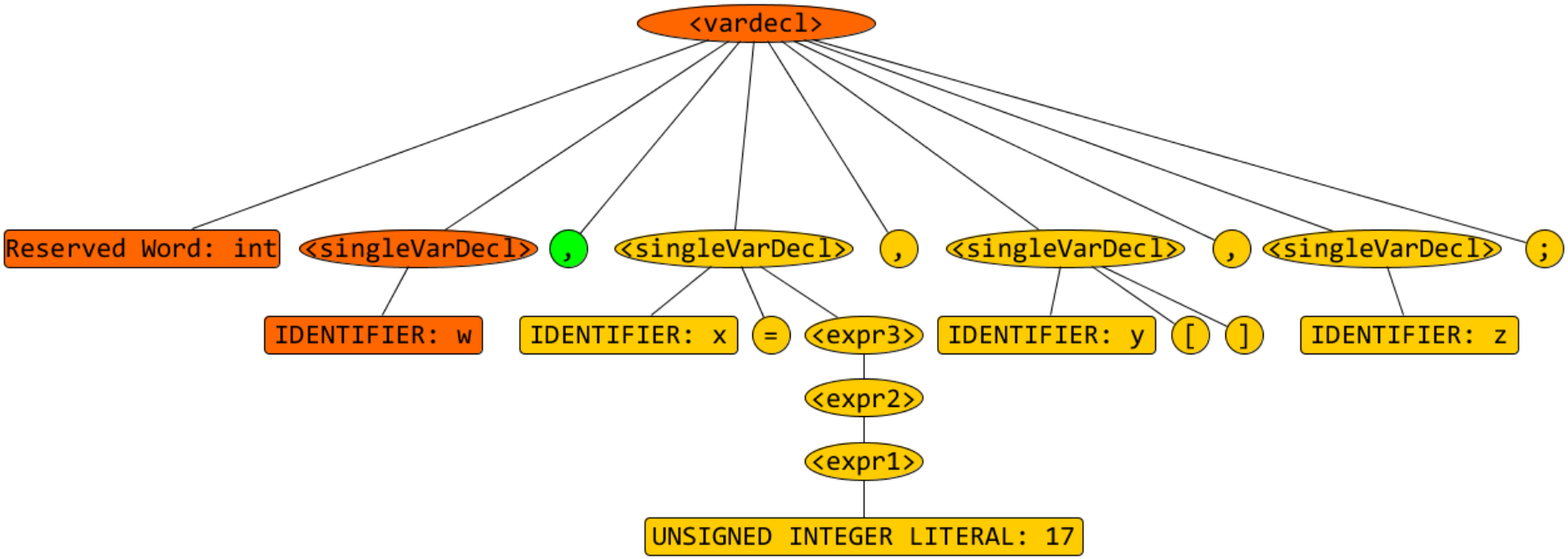
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

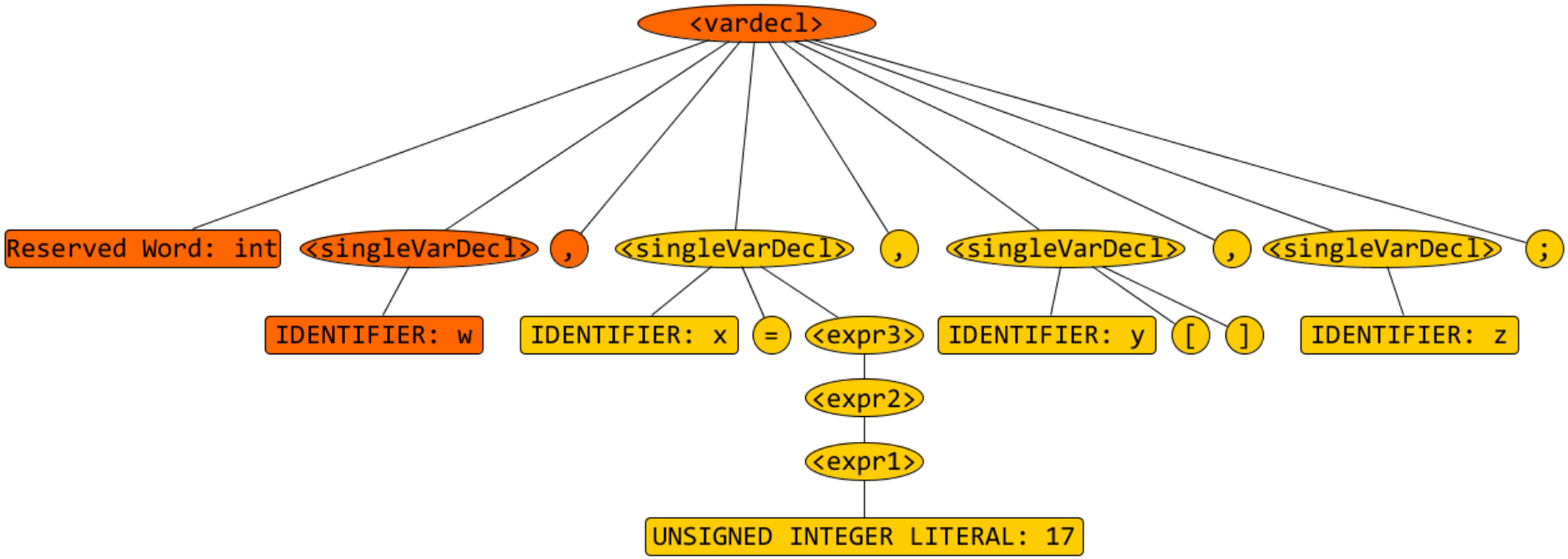
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `nextToken();`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

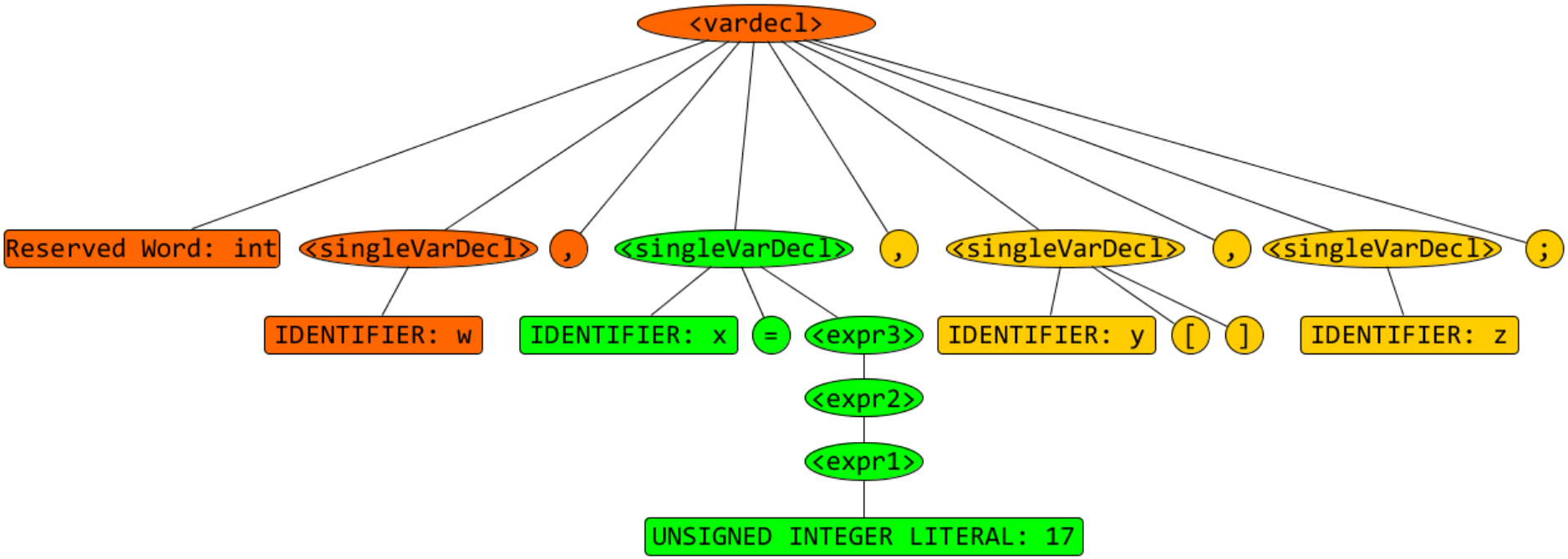
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

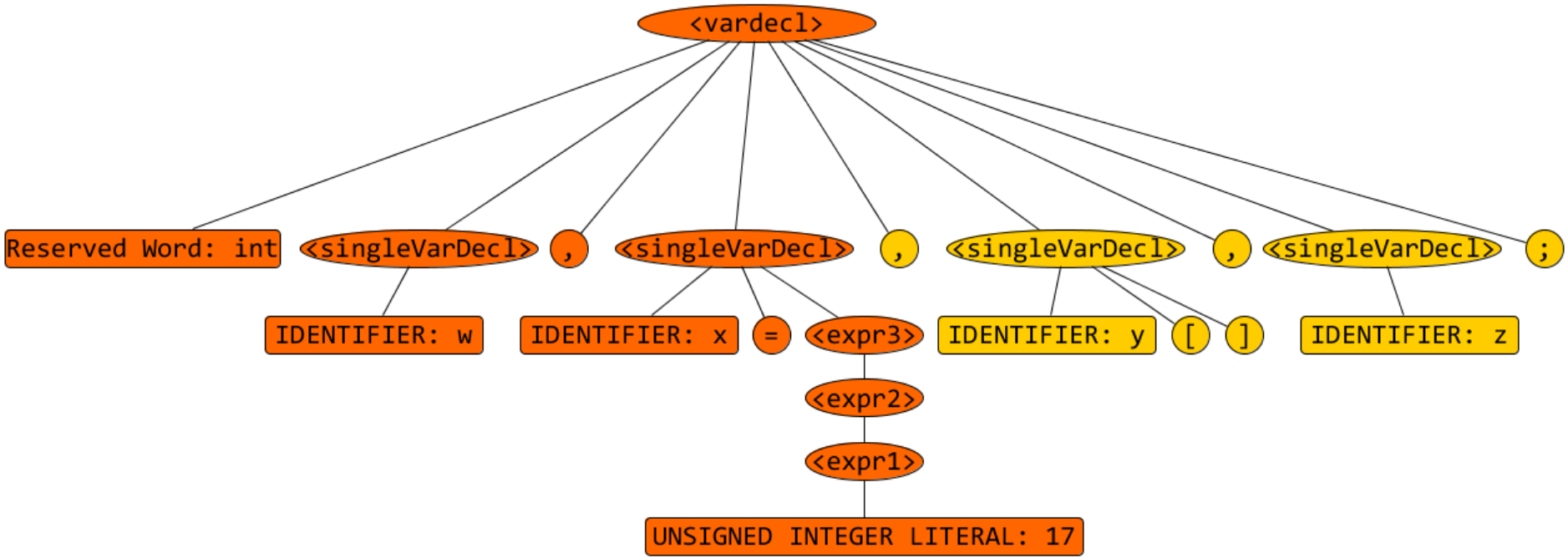
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `singleVarDecl();`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

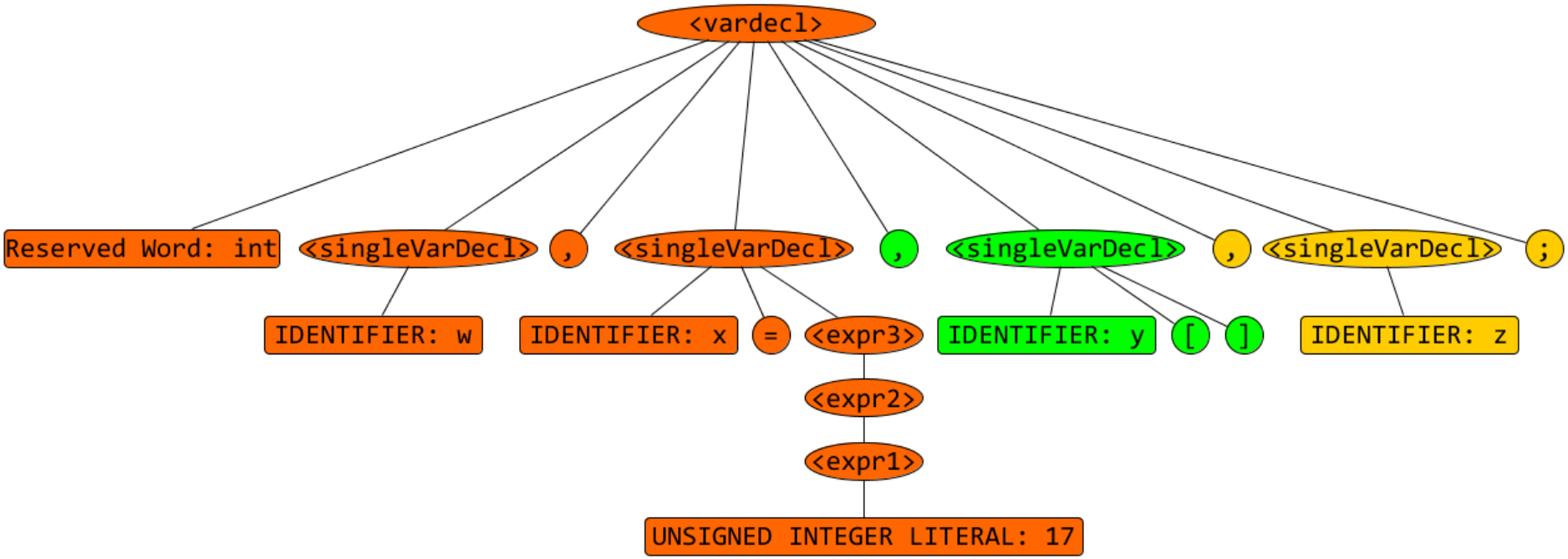
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

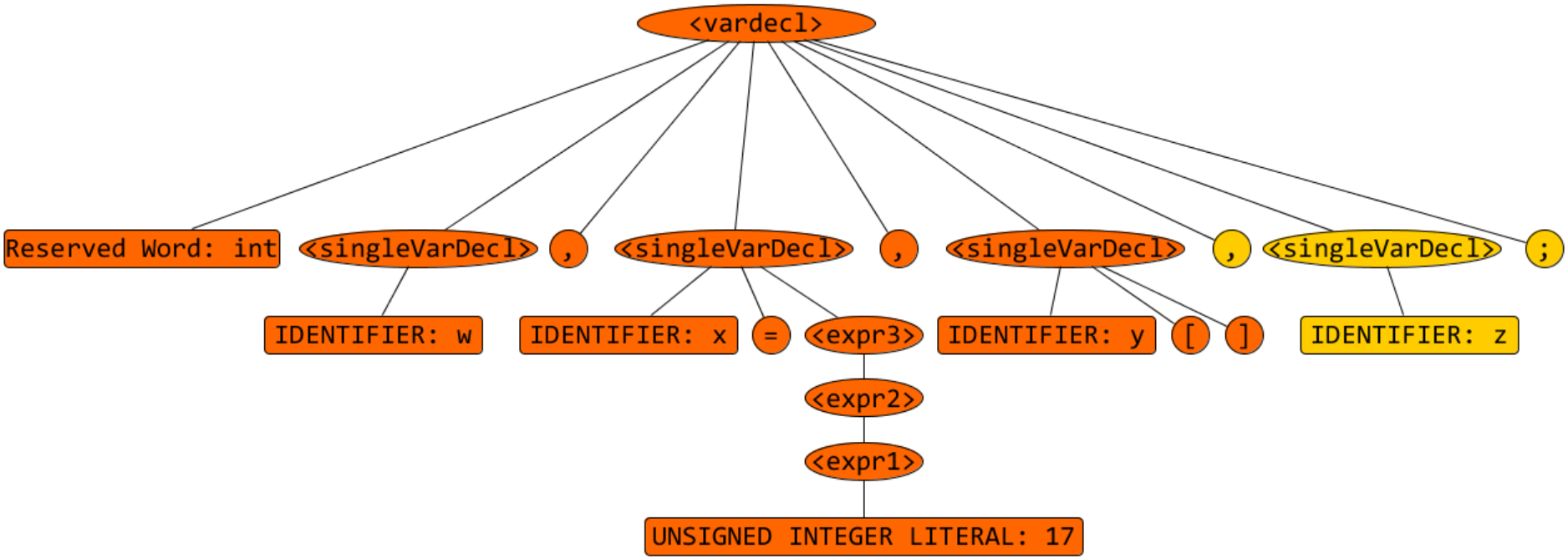
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `nextToken(); singleVarDecl();`

Parse tree of int w, x = 17, y[], z;
with root <varDecl>, based on the following EBNF rule:

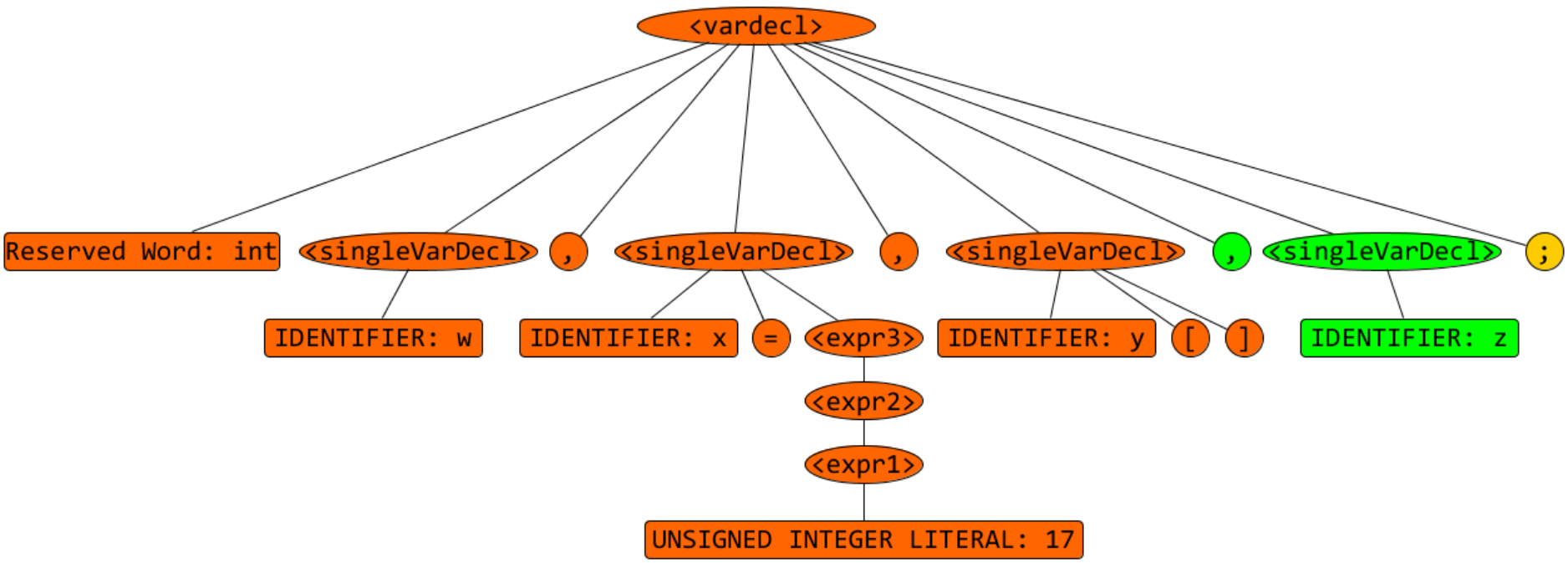
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

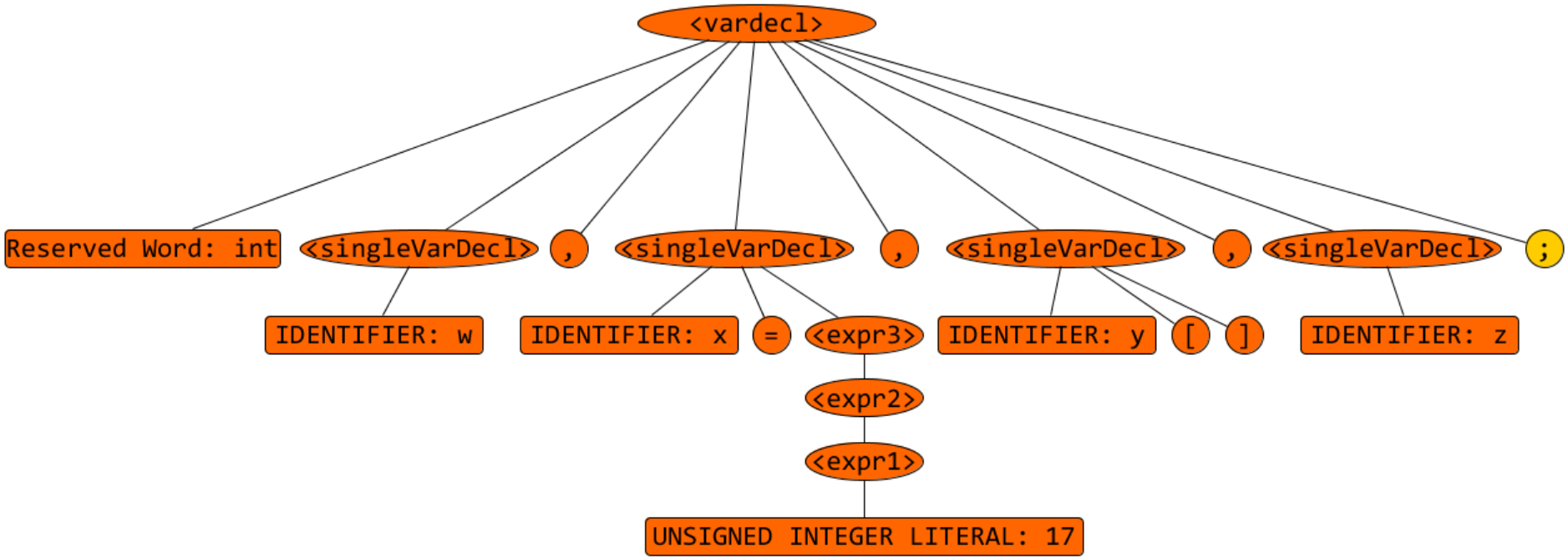
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `nextToken(); singleVarDecl();`

Parse tree of int w, x = 17, y[], z;
with root <varDecl>, based on the following EBNF rule:

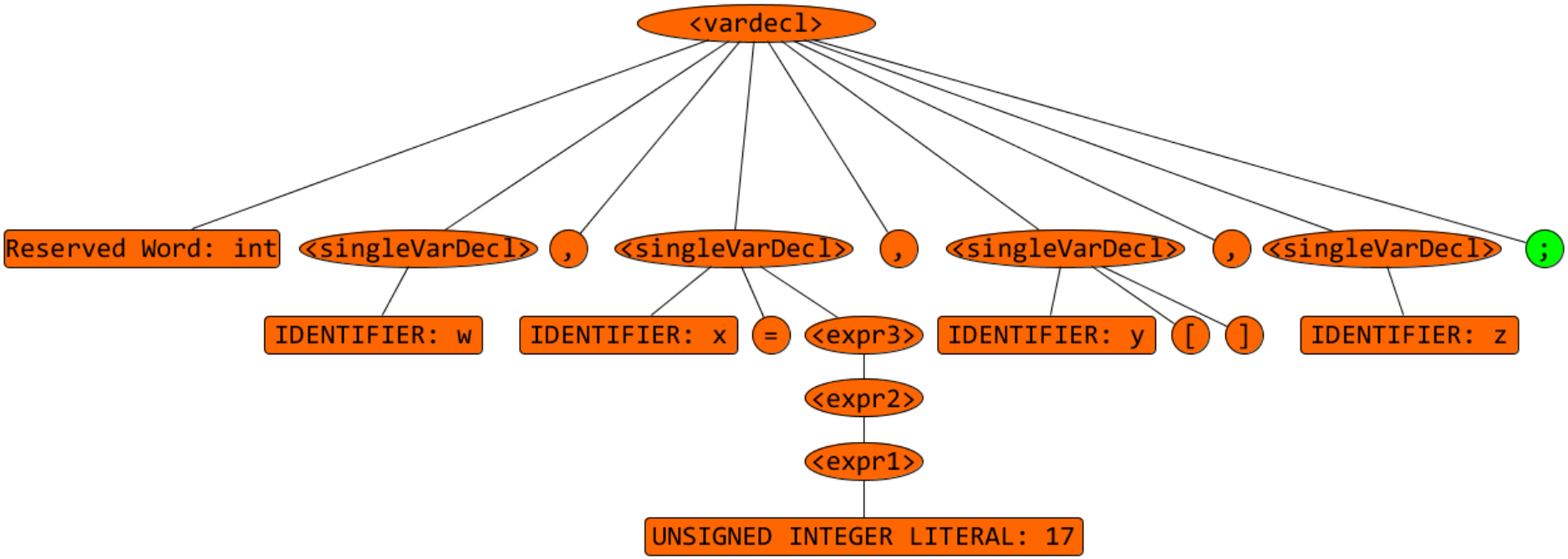
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by:

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

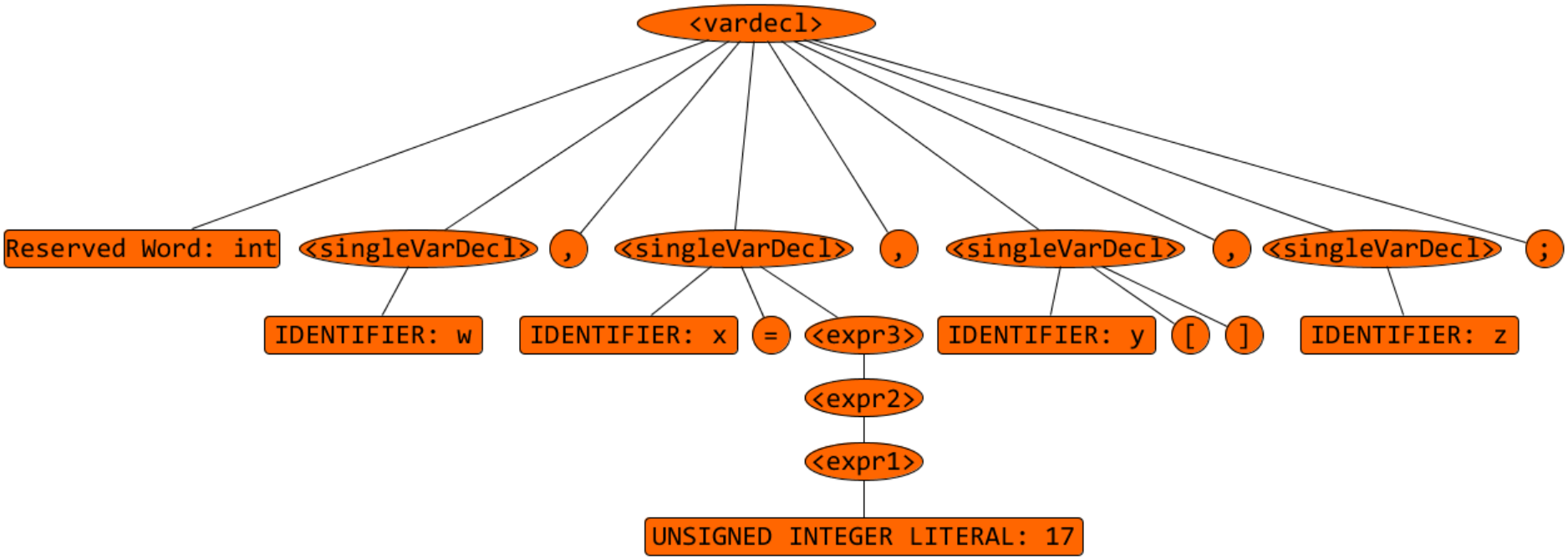
```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



New node(s) created by: `accept(SEMICOLON);`

Parse tree of `int w, x = 17, y[], z;`
with root <varDecl>, based on the following EBNF rule:

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
            |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```



This is the final tree.

```
<varDecl> ::= int <singleVarDecl> { ,  <singleVarDecl>} ;
           |  Scanner IDENTIFIER = new Scanner '('  System . in  ')' ;
```

```
private static void varDecl() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTvarDecl); TJ.output.incTreeDepth();

  if (getCurrentToken() == INT) {
    nextToken();
    singleVarDecl();
    while (getCurrentToken() == COMMA) {
      nextToken();
      singleVarDecl();
    }
    accept(SEMICOLON);
  }
  else if (getCurrentToken() == SCANNER) {
    nextToken();

    if (getCurrentToken() == IDENT) nextToken();
    else throw new SourceFileErrorException("Scanner name expected");

    accept(BECOMES); accept(NEW); accept(SCANNER);
    accept(LPAREN); accept(SYSTEM); accept(DOT);
    accept(IN);  accept(RPAREN); accept(SEMICOLON);
  }
  else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");

  TJ.output.decTreeDepth();
}
```