CPS 2000
COMPILER THEORY AND PRACTICE
REPORT PART 2

ID: 328400L

Name: Ryan Camilleri

Course: Bs.C IT A.I. Second Year

Contents

1. Extending SmallLang

To cater for the addition of the primitive type ‘char’ and for supporting arrays within the SmallLang language, new rules were created and some previous were modified address these changes. These rules are further shown and explained in the following sub-sections.

1.1 Char Rules

To support the inclusion of the new primitive type ‘char’, the *CharLiteral* rule was devised to represent a character value:

<CharLiteral> ::= ‘ \‘ ‘ ∑ ‘ \’ ‘

As shown, a character literal value is comprised of a character surrounded by single quotes. These single quotes are essential for determining whether the character being read is an identifier or a char value. Also, the symbol ∑ denotes all single characters that could be taken as character values. In the lexer implementation (Section 2.1), the allowed alphabet consists of any single characters that could produce a transition within the DFA. All such characters are allowed except for the \n, EOF, and \’ (single quote) characters. The allowed characters are the following:

_ | [A-Z] | [a-z] | [0-9] | . | + | - | * | / | = | > | < | { | } | (|) | , | ; | : | [|]

Modifications to pre-existing rules were also done to cater for the new char type. The *Type* rule was modified to include the char type and the *Literal* rule was modified to include the new literal *CharLiteral*. The two rules can be seen below:

<Type> ::= ‘float’ | ‘int’ | ‘bool’ | ‘char’

<Literal> ::= <BooleanLiteral>
| <IntegerLiteral>
| <FloatLiteral>
| <CharLiteral>

1.2 Array Rules

For including support for arrays, rules which address declaration and assignment were created. The declaration rule *ArrayDecl*, was implemented to have an identifier and type, similar to those of a variable declaration. The rule also has a choice between two types of array declarations. The first type of declaration is when the size of the array is initialised through its indexing operator. In this case, the programmer would have the option of initialising the array immediately with the list of elements or otherwise. The second type, however, deals with the scenario when the size of the array is left empty. In this case, it is necessary for the programmer to assign a list of elements, so that its size could be derived from the size of the list. The rule is as follows:

<ArrayDecl> ::= ‘let’ <Identifier> ‘[‘ <IntegerLiteral> ‘]’ ‘:’ (<Type> | <Auto>) [‘=’ <ArrayElements>]
| ‘let’ <Identifier> ‘[‘ ‘]’ ‘:’ (<Type> | <Auto>) ‘=’ <ArrayElements>

<ArrayElements> ::= ‘{‘ <Expression> { ‘,’ <Expression> } ‘}’

As shown, *ArrayDecl* consists of the above *ArrayElements* rule, which was created to represent the list of elements being initialised. The list could potentially contain an infinite number of elements derived from expressions. This was done to give the programmer the option of initialising the array with results from binary operations or factors, if required. Even though the list contains may have a greater/smaller number of elements than the size of the array, no errors are raised when checking for validity of the program. This, however, would eventually be checked for semantic correctness, which will raise an error when comparing the two sizes (not tackled within this assignment). Another rule *ArrayAssignment* was created to cater for element assignment at the array position specified by an index. The rule is as follows:

<ArrayAssignment> ::= <Identifier> ‘[‘ <IntegerLiteral> ‘]’ ‘=’ <Expression>

As shown, an identifier is required for using the right array which would be inserted in the symbol table. Also, an *IntegerLiteral* is also required to specify the index which will be initialised with the value from *Expression*. To account for the creation of the described statements, the *Statement* rule was modified to include both, as shown:

<Statement> ::= <ArrayDecl> ‘;’
| <ArrayAssignment> ‘;’
| …

To retrieve elements from arrays, the rule *ArrayCall* was created. This rule represents how a typical array call would be structured whereby the array specified by the identifier is used to return the element value, at the specified position. This rule is similar to *FunctionCall* which returns a value to be used within expressions. Thus, the *Factor* rule was modified to include *ArrayCall* as shown:

<ArrayCall> ::= <Identifier> ‘[‘ <IntegerLiteral> ‘]’

<Factor> ::= <ArrayCall>
| <Literal>
| …

Lastly, function declarations were also given the ability to have array formal parameters. This was achieved by modifying the *FormalParam* rule to have an optional indexing operator as shown:

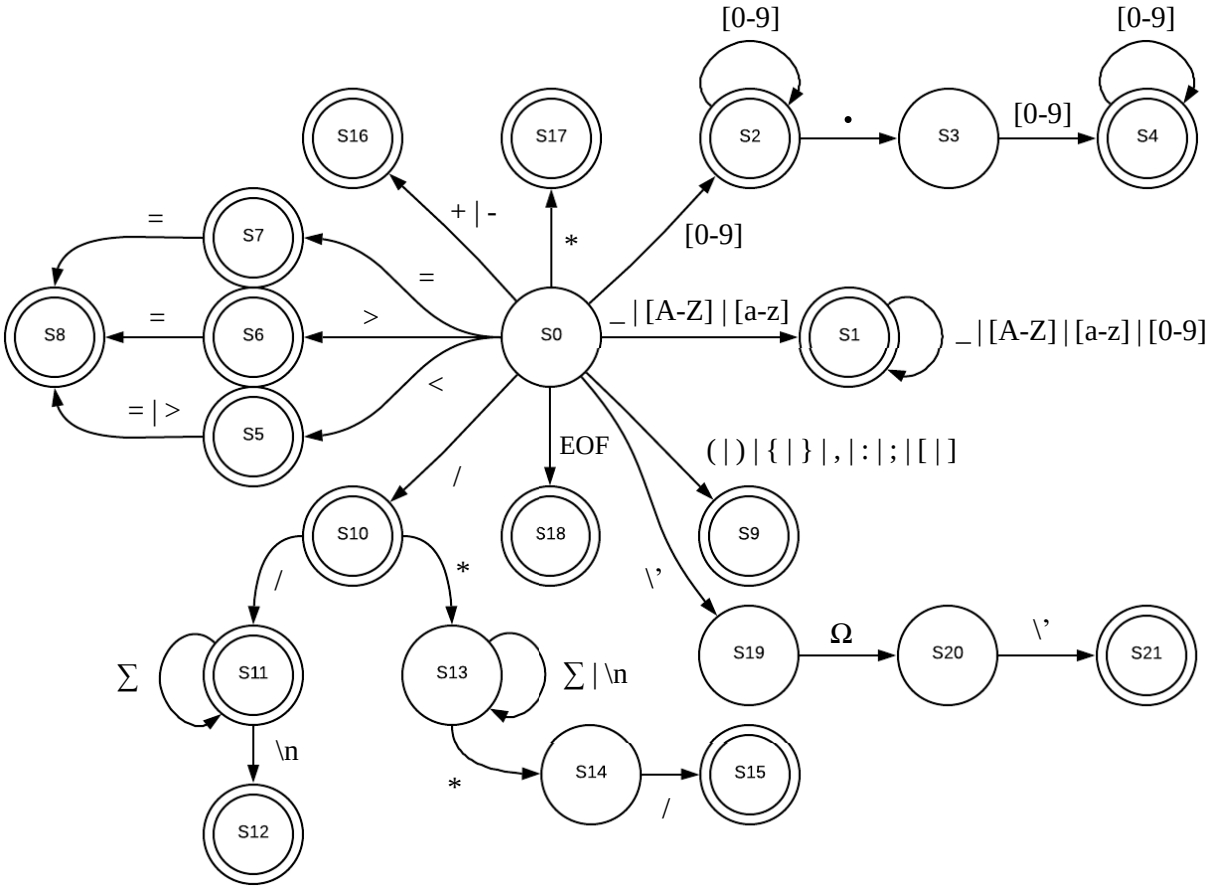
<FormalParam> ::= <Identifier> [‘[‘ ‘]’] ‘:’ <Type>

2. SmallLangV2 Lexer and Parser

In order to implement the SmallLangV2 extensions, the lexer and hand-crafted parser were altered to handle such change. The modifications done on each is further explained in the following sub-sections.

2.1 Lexer

Firstly, a new token type *TOK_CharLit* was added in the enumerator storing all token types called *TokenType*. This token was created to represent the character literal rule, which will be returned by a specific state within the DFA. That being said, the automaton was also modified to handle the character and array extensions as shown below:



State	Token Type
S19	TOK_Error
S20	TOK_Error
S21	TOK_CharLit

Table 2.1.1: New State and Token Type Table

Three new states, S19, S20 and S21 were added to the DFA. These states were created specifically for handling character values which are to be surrounded by single comments. This was done since, given that state S1 was used to find characters, extra checks would need to be done within the lexer itself to determine whether the value is an identifier or a character value. With the inclusion of these states however, the value is immediately determined since every character surrounded by single comments can be defaulted to a char literal, returned by final state 21.

It should be noted that the symbol **Ω** on the transition between S19 and S20, refers to all possible character values that raise a transition within the DFA, except for the \n, EOF and \' characters. Another modification was done for the state S9, so that it could now cater for square bracket punctuation for an array's indexing operator. The newly defined transition table which can be seen below, includes the newly added states along with the new alphabet symbol ' \' ' used to determine character values.

	α	[0-9]	.	+ | -	*	/	=	>	<	β	\n	EOF	\'
S0	1	2	-1	16	17	10	7	6	5	9	0	18	19
S1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S2	-1	2	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S3	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S4	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S5	-1	-1	-1	-1	-1	-1	8	8	-1	-1	-1	-1	-1
S6	-1	-1	-1	-1	-1	-1	8	-1	-1	-1	-1	-1	-1
S7	-1	-1	-1	-1	-1	-1	8	-1	-1	-1	-1	-1	-1
S8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S10	-1	-1	-1	-1	13	11	-1	-1	-1	-1	-1	-1	-1
S11	11	11	11	11	11	11	11	11	11	11	12	-1	11
S12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S13	13	13	13	13	14	13	13	13	13	13	13	-1	13
S14	-1	-1	-1	-1	-1	15	-1	-1	-1	-1	-1	-1	-1
S15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S19	20	20	20	20	20	20	20	20	20	20	-1	-1	-1
S20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	21
S21	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 2.1.2: Transition Table

Where α is _ | [A-Z] | [a-z] and β is (|) | { | } | , | : | ; | [|]

Within the *Lexer* class itself, the 'char' keyword was also added when checking for the final state S1. In this case, given that the lexeme would accumulate to a string "char", this would be identified as a keyword, and the state would return a token of type *TOK_Type* with lexeme "char".

2.2 Parser

When extending the parser, more AST nodes had to be created for addressing the new EBNF rules shown in section 1. The *Parser* class was also modified to include parsing functions to handle such rules. By doing so, other pre-existing functions had to be modified to cater for this change. These are all explained in detail, in the following sub-sections.

2.2.1 AST Nodes

Firstly, the enumerator called *Type* was appended with the CHAR value to handle character types. An AST node *ASTCharLiteral*, was also created to store a character value. The node *ASTType* was also modified to check if the passed lexeme is equal to "char". If so, the type stored in the node would be initialised to the value CHAR denoted by the enum *Type*. By doing these changes, the primitive char type was completely addressed in terms of AST structure and support.

For handling array data structures, the two statement nodes *ASTArrayDecl* and *ASTArrayAssignment* were built. Both of these nodes extend the abstract class *ASTStatementNode* and are organised within the *Statements* directory. *ASTArrayDecl* was defined to have variables of type *ASTIdentifier, ASTIntegerLiteral, ASTType,* and *ASTArrayElements*, which are all necessary on array declaration. The *ASTArrayElements* node is another node which was created solely for array support. Such node stores a list of expression nodes to be used as element values on declaration. *ASTArrayAssignment* also includes variables of type *ASTIdentifier* and *ASTIntegerLiteral* to identify the array being used, and at which position the expression should be stored in. Such expression is stored in a variable of type *ASTExpressionNode*, also within the class itself. Lastly, the expression node *ASTArrayCall* was created, which extends the *ASTExpressionNode* class; organised within the *Factors* directory with variables of type *ASTIdentifier* and *ASTIntegerLiteral*. All new and modified nodes can be viewed in the diagram below:

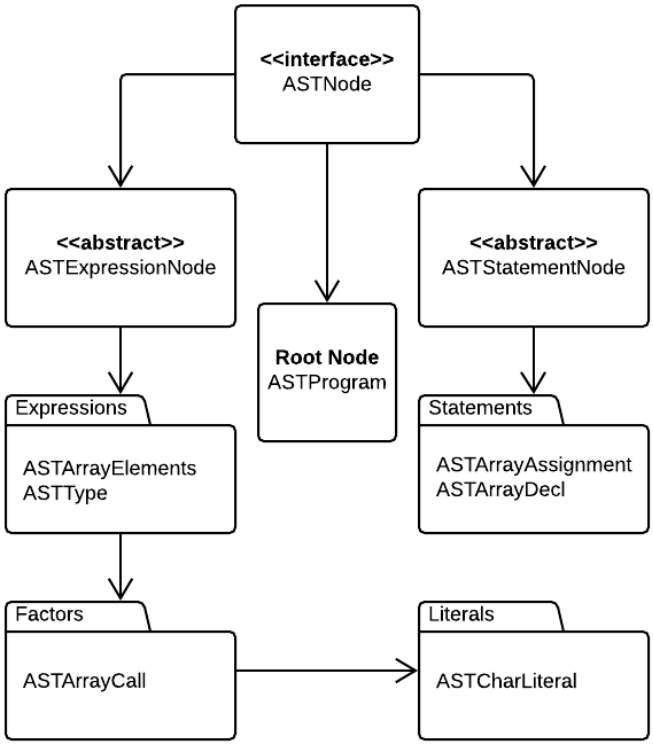


Figure 2.2.1: Directory structure with new and modified AST classes

2.2.2 Parser Modifications

To handle support for both char type and arrays, specific functions were created for parsing, whilst other pre-existing functions were modified for supporting these structures. The new functions implemented within the *Parser* class, include:

- **parseDeclaration**
  This function can be seen as a general function to determine whether the declaration being made is of a variable type or an array type. It manages this process by consuming the identifier token and calls the Lexer look ahead to check the next token following the declaration identifier. Given that the next token is a semicolon ' : ', the *parseVariableDeclaration* is called. If however, the token is an open square bracket ' [' (beginning of an indexing operator), the *parseArrayDeclaration* function is called instead. In each case, the identifier consumed at the beginning, is passed to the called function for it to be used within them. If the lookahead is not a semicolon nor an open square bracket, an error is displayed, and the program is exited.

- **parseVariableDeclaration**
  Even though a similar function already existed in the previous version of the parser class, another overloaded function was created to accept an identifier from *parseDeclaration*. The same process is carried out, whereby a new *ASTVariableDecl* node is returned at the end, using the passed identifier (given that no errors occur during parsing of the function).

- **parseArrayDeclaration**
  Function used to parse an array declaration statement. It is responsible for carrying out the necessary checks determining the validity of the statement whilst also checking the type of array declaration. It does this by ensuring that if no array size was initialised within the program, an elements list must exist for the size to be determined from it. If this was not the case and both the size of the array and the elements list weren't included, an error is shown, and the program is exited.

- **parseArrayElements**
  Function called by *parseArrayDeclaration* given that an elements list was assigned to the declared array. This function is responsible for parsing the list of elements, storing the expression nodes within a list, and returning an *ASTArrayElements* node with this list, if no errors were raised.

- **parseArrayAssignment**
  Function used to ensure that the array assignment done, is valid. If all the necessary values were programmed, an *ASTArrayAssignment* node is returned by the function with a valid identifier, index, and expression that the array was assigned to.

- **parseArrayCall**
  Function used to ensure that the array call done, is valid. An *ASTArrayCall* node is returned with a valid identifier and index, given that no errors were raised during parsing of the factor-expression.

The pre-existing modified function include:

- **parseStatement**
  The function was modified to handle two new scenarios, when choosing which declaration to parse, and when choosing which assignment to parse. In the case for declarations, the choice had to be done whenever the current token has a "let" value. In this case, the function has to determine whether to call *parseVariableDeclaration* or *parseArrayDeclaration*. Due to the parser being an LL(1) parser, the lookahead could only show the identifier of the statement and not what follows after it. Thus, the *parseDeclaration* function (explained previously) was

created to be called whenever the “let” keyword is present. By doing so, instead of defining logic within the *parseStatement* function itself, logic to choose the proper declaration function was carried out in *parseDeclaration* by consuming the identifier and performing lookahead of the token following after it.

In the case for assignment statements, a lookahead is performed within *parseStatement* since this would identify whether the identifier being assigned, is followed by an indexing operator, or otherwise. If it is followed by the operator, the *parseArrayAssignment* function is called, otherwise *parseAssignment* is called.

- **parseFactor**
  This function was modified to include checks for supporting array calls and character literals. For the case when the current token is of type *TOK_CharLit*, an *ASTCharLiteral* node is returned with the character value without the single quotes which previously surrounded it. When the token if of type *TOK_Identfier* however, an extra lookahead is performed to check if the next token is the beginning of an array’s indexing operator. If it is, the *parseArrayCall* function is called, and its value returned by *parseFactor*.

- **parseFormalParam**
  The function was modified to cater for its rule modification shown in section 1. In the case where the formal identifier is followed by an open square bracket, it is ensured that the syntax of the indexing operator is valid or not. If, however, the first character of the indexing operator isn’t detected, the previous parsing process is carried out instead.

3. Tool Generated Parsers

For this part of the assignment, research had to be done on which compiler-compiler tool was to be used for generating a parser for the SmallLang grammar. This parser would replace the hand-crafted parser done in the first assignment, which generates a parse tree instead. The structure of such tree is dependent on the grammar used, which will be hand-written to best represent the SmallLang languages.

3.1 Compiler-compiler tools

Due to ANTLR being one of the most widely used and most recommended parser generators out there, it was decided that this would be the tool to be used for parser generation. As described in [1], the author who's also the developer of ANTLR, states how the software is based on a new LL algorithm which uses rules defined within the grammar to parse the input program and generate the parse tree describing it. ANTLR also provides a number of extra tools, namely visitors which interact with the elements of the tree. This will be further described in section 4 of the assignment. After research on how to build a proper ANTLR grammar and best practices involved when constructing one was completed, work was shifted on adopting ANTLR within my coding environment. This was achieved by installing the latest ANTLR plugin in my IntelliJ directory (IDE being used) for allowing '.g4' grammar files and by including the *antlr-4.8-complete* jar file (installed from the official ANTLR website [2]) in the environment's dependencies. By including this file, all artefacts generated by building the grammar would have access to ANTLR's runtime environment, and so, is essential.

3.2 Grammars

Each and every ANTLR grammar is split into lexer rules and parser rules. The lexer rules which start with an upper case character, define almost every EBNF rule that was tackled by the DFA implementation, using tokens. The parser rules on the other hand, all start with a lowercase character and describe all expression and statement rules present in the language's EBNF. These rules can be divided to separate files, however, for my implementation of the grammar, the rules were combined into one grammar '.g4' file. Two grammars were constructed using aid from a cheat sheet [3] describing the different symbols and their use. The first grammar is called *SmallLang.g4* and the second *SmallLangV2.g4*. The second is an extension of the first and has all the same rules except for those that were modified and added to the second language. Each grammar generates grammar-specific artefacts (like a base lexer and a base parser) which are stored in a specific directory within the *src* directory. The directory structure solely related to ANTLR, is the following:

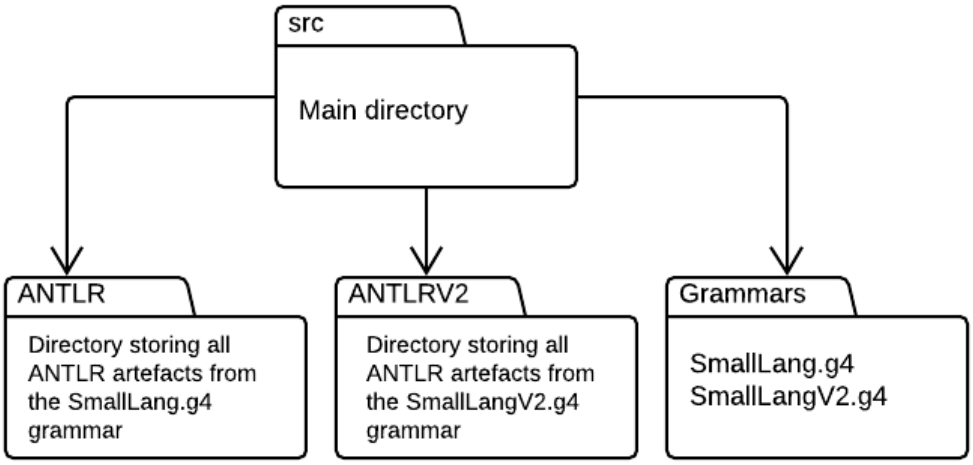


Figure 3.2.1: Directory structure for ANTLR directories

3.2.1 Lexer Rules

Each constructed rule represents closely those found in the EBNF specification for part 1 of the compiler assignment. The following lexer rules were defined for the SmallLang grammar:

```
fragment LETTER : [a-zA-Z];
fragment DIGIT  : [0-9];

TYPE            : 'float' | 'int' | 'bool';
AUTO            : 'auto';
BOOLEANLITERAL  : 'true' | 'false';
INTEGERLITERAL  : DIGIT+;
FLOATLITERAL    : DIGIT+'.'DIGIT+;
MULTIPLICATIVEOP: '*' | '/' | 'and';
ADDITIVEOP      : '+' | MINUS | 'or';
RELATIONALOP    : '<' | '>' | '==' | '<>' | '<=' | '>=';
UNARYOP         : MINUS | 'not';

//Put under all others since a term like 'and' would be identified as an
identifier first instead of mult operator
IDENTIFIER      : ( '_' | LETTER ) ( '_' | LETTER | DIGIT )*;

//Extra-lexer Rules for above rules
//Done to eliminate risk of not detecting non-referenced elements
MINUS           : '-';

//Characters and comments to ignore
NEWLINE         : ('\r'? '\n' | '\r')+ -> skip;
WHITESPACE      : (' ' | '\t') -> skip;
MULTICOMMENT    : '/*' .*? '*/' -> skip;
COMMENT         : '//' ~( '\n' )* -> skip;
```

As shown the lexer rules are further split into fragments and non-fragments. Fragment rules are those which can be used within other lexer rules of a non-fragment type. Example being the fragment *DIGIT* used in the *INTEGERLITERAL* rule. It was observed that rule precedence is of utmost importance when keywords like “int”, “and” and more were not being recognised as their specified rule type, but instead being specified as IDENTIFIER. This was due to the fact that the IDENTIFIER rule was initially placed at the beginning of the non-fragment rule list. Thus, when a lexeme like “int” was to be identified, the token returned by the lexer was of type IDENTIFIER, since this was satisfied before the TYPE token. By fixing this issue, every rule is checked for any matching keywords and if none are satisfied, the lexeme input is defaulted to an IDENTIFIER.

For some odd reason, when observing the generated parse tree, an error was being shown when the minus ‘-‘ additive operator was used. All lexer and parser rules were checked for correctness to determine if any syntactical errors existed, but none were found. The only solution that lead for the minus operator to be recognised, was by including the rule MINUS, which is referenced within the ADDITIVEOP and UNARYOP rule. By doing such reference, the parser could correctly identify the operator. UNARYOP was also included as a new rule to be used within the *unary* parser rule. Lastly, rules to address whitespaces, newlines, and comments were created. These rules were appended with the ‘-> skip’ notation for them to be ignored by the lexer, when encountered.

3.2.2 Parser Rules

Similar to the lexer rules defined, each parser rule was built having a very similar structure to the list of EBNF rules for the language. The rules defined are the following:

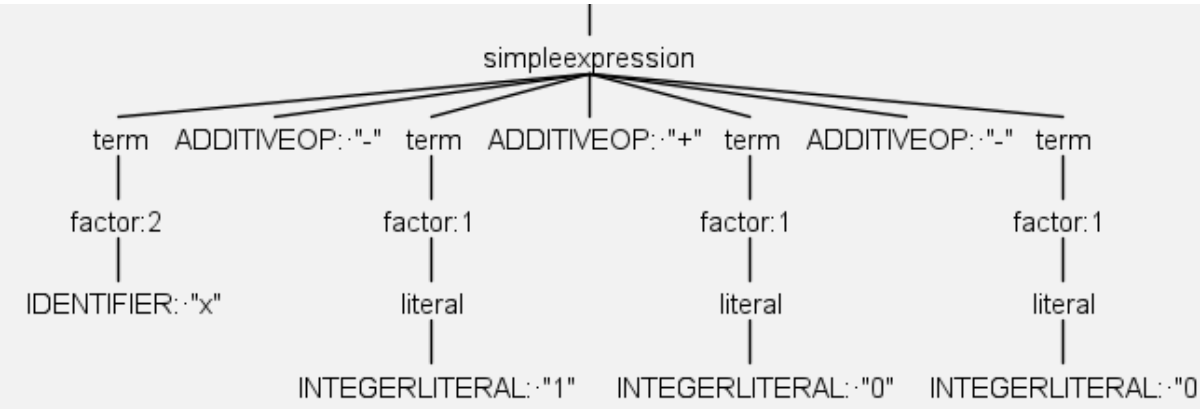
```
//Expressions
literal          : BOOLEANLITERAL
                 | INTEGERLITERAL
                 | FLOATLITERAL;

actualparams     : expression (',' expression)*;
functioncall     : IDENTIFIER '(' actualparams? ')';
subexpression    : '(' expression ')';
unary            : UNARYOP expression;

factor           : literal
                 | IDENTIFIER
                 | functioncall
                 | subexpression
                 | unary;

term             : factor MULTIPLICATIVEOP factor
                 | factor MULTIPLICATIVEOP term
                 | factor;

simpleexpression: term ADDITIVEOP term
                 | term ADDITIVEOP simpleexpression
                 | term;

expression       : simpleexpression RELATIONALOP simpleexpression
                 | simpleexpression RELATIONALOP expression
                 | simpleexpression;

//Statements
assignment       : IDENTIFIER '=' expression;
variabledecl     : 'let' IDENTIFIER ':' (TYPE | AUTO) '=' expression;
printstatement   : 'print' expression;
rtrnstatement    : 'return' expression;
ifstatement      : 'if' '(' expression ')' block ('else' block)?;
forstatement     : 'for' '(' (variabledecl)? ';' expression ';' (assignment)?
                 ')' block;
whilestatement   : 'while' '(' expression ')' block;
formalparam      : IDENTIFIER ':' TYPE;
formalparams     : formalparam (',' formalparam)*;
functiondecl     : 'ff' IDENTIFIER '(' formalparams? ')' ':' (TYPE | AUTO)
                 block;

statement        : variabledecl ';'
                 | assignment ';'
                 | printstatement ';'
                 | ifstatement
                 | forstatement
                 | whilestatement
                 | rtrnstatement ';'
                 | functiondecl
                 | block;

block            : '{' (statement)* '}';
program          : (statement)*;
```

As shown, each rule defined is built from other rules, both from the lexer and the parser itself. Rules like that of whilestatement, also define the required syntax for a valid while loop. That is, it requires the keyword, "while", followed by an open bracket "(", an *expression* rule, a closed bracket ")" and lastly the *block* rule. If all these conditions are satisfied from the input program, the while statement would be defined for such tokens and the parse tree would build a while statement node with this information. In some instances, rule can consist of sub-rules followed by the symbol '*'. This would mean that the sub-rule could occur zero or more times within the rule. An example where this is used is within the *block* rule, which can have 0 or more statements. In other instances, sub-rules may also be followed by the '?' symbol. This symbol denotes the fact that the sub-rule is optional and that it can be included or disregarded. An example of this is used within the *functiondecl* rule, which has an optional *formalparams* sub-rule (function may or may not have formal parameters).

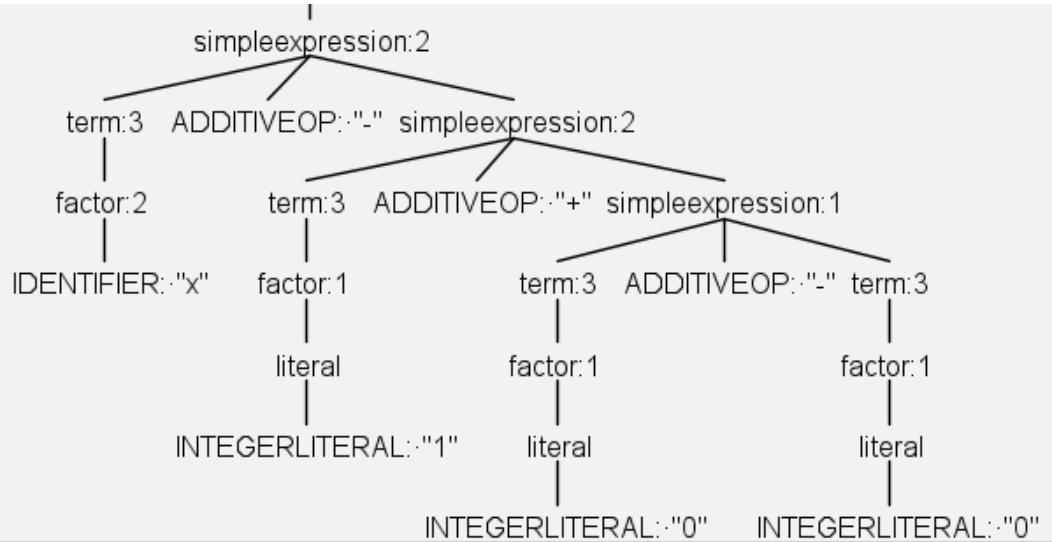
In the first implementation of the parser rules, the rules *term*, *simpleexpression* and *expression* were implemented differently. The structure of how these were implemented is shown below:

```
term           : factor (MULTIPLICATIVEOP factor)*;
simpleexpression: term (ADDITIVEOP term)*;
expression     : simpleexpression (RELATIONALOP simpleexpression)*;
```

The above rules closely resemble those found in the EBNF specification, whereby a simple expression for example, may have a single term or have a left term, followed by multiple additive operators and other terms. The problem with this definition was found when trying to convert the produced parse tree to its AST equivalent, for task 4 of the assignment. In such cases, where an AST node had to be created for each of the above expressions, recursion couldn't be carried out. This was due to the fact that when multiple operators and expressions existed, the *simpleexpression* node (for example) within the parse tree would have a list of children in the order they occurred in the input program. The below diagram shows an example of this:



As it stood, the *simpleexpression* node shown, consisted of no other *simpleexpression* contexts within its children, and so, couldn't be recursed. This implementation was first resolved by catering for the issue by building an iterative solution to imitate the recursive procedure and grouping terms together as simple expressions from right to left until the leftmost term was reached. However, when searching into grammar recursion for the ANTLR kit, the parser rules were modified so that left-recursion could be adopted. By doing so, the above tree structure would look like so:



As shown, by modifying the grammar rules to how they were displayed in the beginning of this section, the structure of nodes for expressions, would look like the one above. By doing this change, recursion could be used when visiting the context of the right expression, making life much easier. The importance of precedence was also applied to these expression rules.

```
simpleexpression: term ADDITIVEOP term
                | term ADDITIVEOP simpleexpression
                | term;
```

Taking the above example, the option `term ADDITIVEOP simpleexpression`, was initially applied before the `term ADDITIVEOP term` option. As explained before, an ANTLR grammar is sensitive to precedence and thus, when the two options were shifted, the parse tree displayed an incorrect structure. Given that the input would consist of just the left term, operator and the right term, the right term would have a parent *simpleexpression* node, since the first option would be traversed and satisfied first. Thus, to fix this issue, the correct order at which options are tested was applied, and the parse tree then outputted a correct structure.

3.2.3 SmallLangV2 Extensions

For addressing character and array support, the following rules were created/modified:

```
TYPE                : 'float' | 'int' | 'bool' | 'char';
CHARLITERAL         : '\'' (LETTER | DIGIT | '_') '\'';
```

The above two lexer rules were created to support the char primitive type. As shown, CHARLITERAL can have a character value of LETTER, DIGIT or '_' only. This isn't the same as the one implemented in the previous tasks of the assignment; however, this was kept as such for simplicity's sake when testing on ANTLR. The following are the parser rules created:

```
arraycall       : IDENTIFIER '[' INTEGERLITERAL ']';


factor          : literal
                | IDENTIFIER
                | functioncall
                | arraycall
                | subexpression
                | unary;


formalparam     : IDENTIFIER ('[' ']')? ':' TYPE;


arraydecl       : 'let' IDENTIFIER '[' (INTEGERLITERAL)? ']' ':' (TYPE | AUTO)
                ( '=' arrayelements )?
                | 'let' IDENTIFIER '[' ']' ':' (TYPE | AUTO) '=' arrayelements;


arrayelements   : '{' (expression (',' expression)* ) '}';

arrayassignment : IDENTIFIER '[' INTEGERLITERAL ']' '=' expression;

statement       : variabledecl ';'
                | assignment ';'
                | printstatement ';'
                | ifstatement
                | forstatement
                | whilestatement
                | rtrnstatement ';'
                | functiondecl
                | block
                | arraydecl ';'
                | arrayassignment ';';
```

As shown, the above parser rules handle array recognition, and some pre-existing rules like *factor* and *formalparam* were modified to support this inclusion.

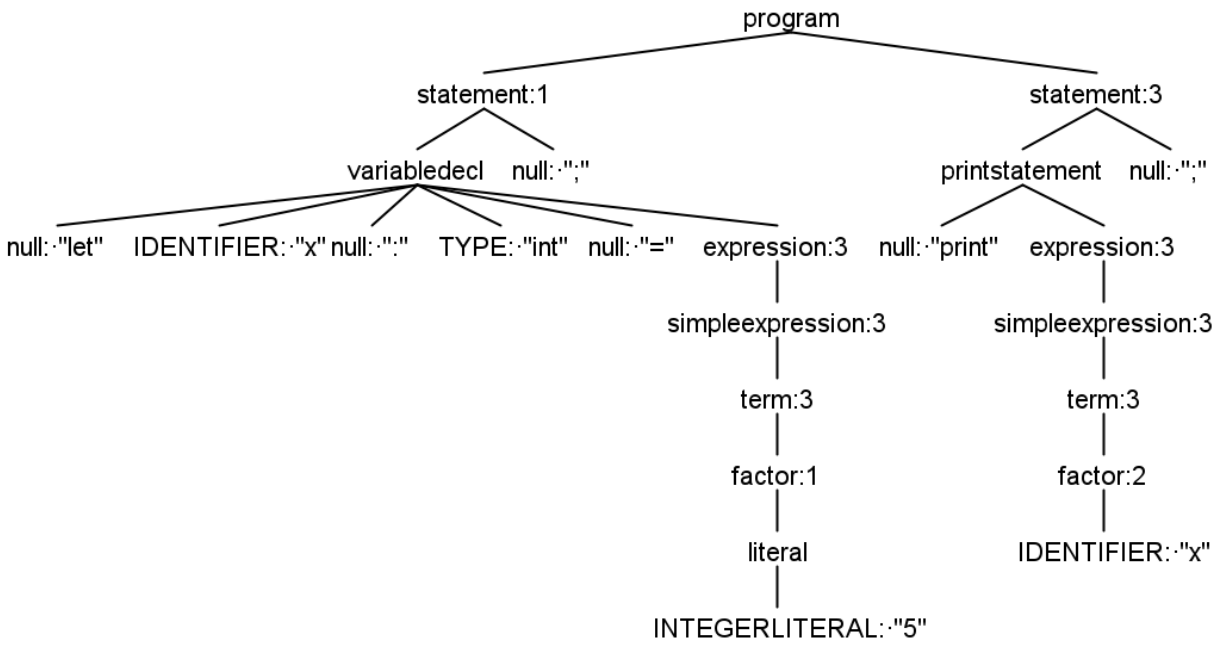
3.3 Parse Trees

The following sub-sections show some parse trees generated by the compiler-compiler parser, when given an input program. All generated parse trees were constructed from the SmallLangV2 grammar since this consisted of all language rules including those of arrays and characters. Moreover, parse tree visualisation was achieved through ANTLR's own tree preview from the IntelliJ plugin installed.

3.3.1 Parse Tree for small program

Due to how large parse trees can grow when given a substantially long input program, the following two-line program was used to display the tree, so that correctness can be easily determined.

```
1   let x:int = 5;
2   print x;
```

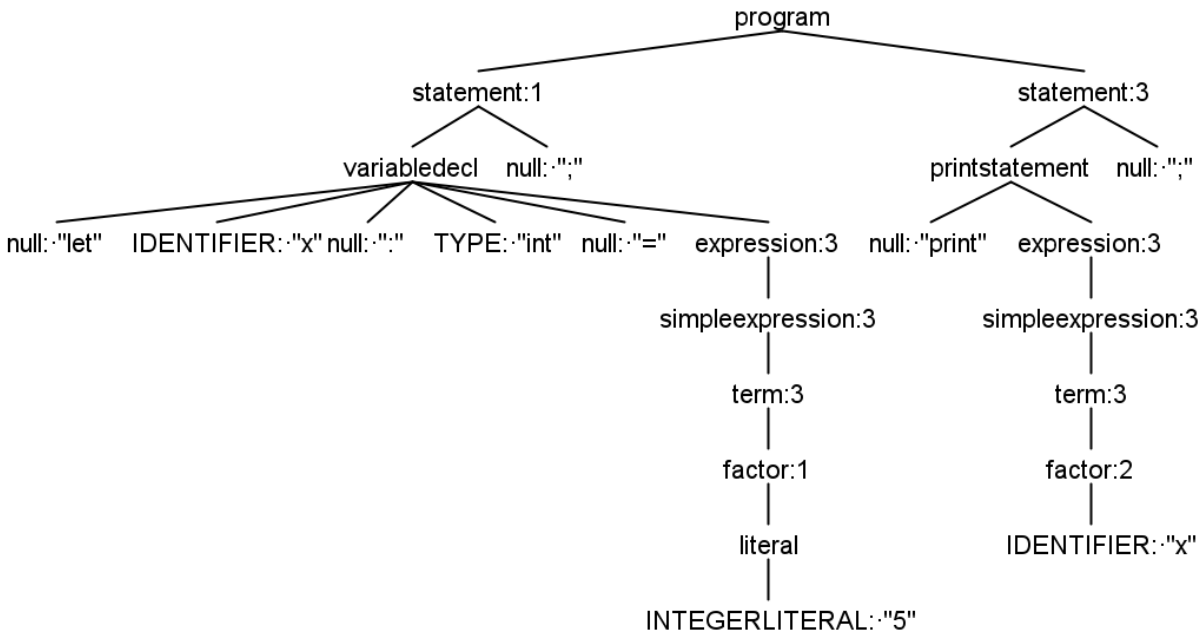


As shown, the above tree is generated from the two-line input. The parse tree is observed to start at the program root node, which is similar to the root node present in the AST equivalent. The program rule should consist of a list of global statements within the program. The above tree, validates this claim since the two statements in the input, are children to the program root node. The first statement should correspond to a variable declaration and the second, should be a print statement. These requirements are also satisfied by the parse tree. It should be noted that whenever a specific syntax was defined in the parser rules, to which no lexer definition exists for it, the symbol is tagged as a null value since this isn't addressed by the lexer. An example of this, is the semicolon ' ; ' following the *variabledecl* statement. When traversing the whole tree, it can be concluded that it is indeed valid and that all tokens were recognised well and assigned to their appropriate rule-tags. The structure is very similar to that of an AST; however, the parse tree stores more child nodes since it deals with null values. The AST data structure doesn't, and instead, for nodes like that of *variabledecl* it would only store the most relevant information which are the nodes of IDENTIFIER, TYPE and *expression*.

3.3.2 Parse Tree for commented program

Using the same input program as the one used in the previous example, comments were added to the program to determine if any changes occurred within the parse tree. As shown, the below input makes use of both inline comments and multi-line comments.

```
1   /* This
2   is
3   a
4   multi-line
5   comment */
6
7   let x:int = 5;
8
9   //Printing x
10  print x; //Value of 5
```

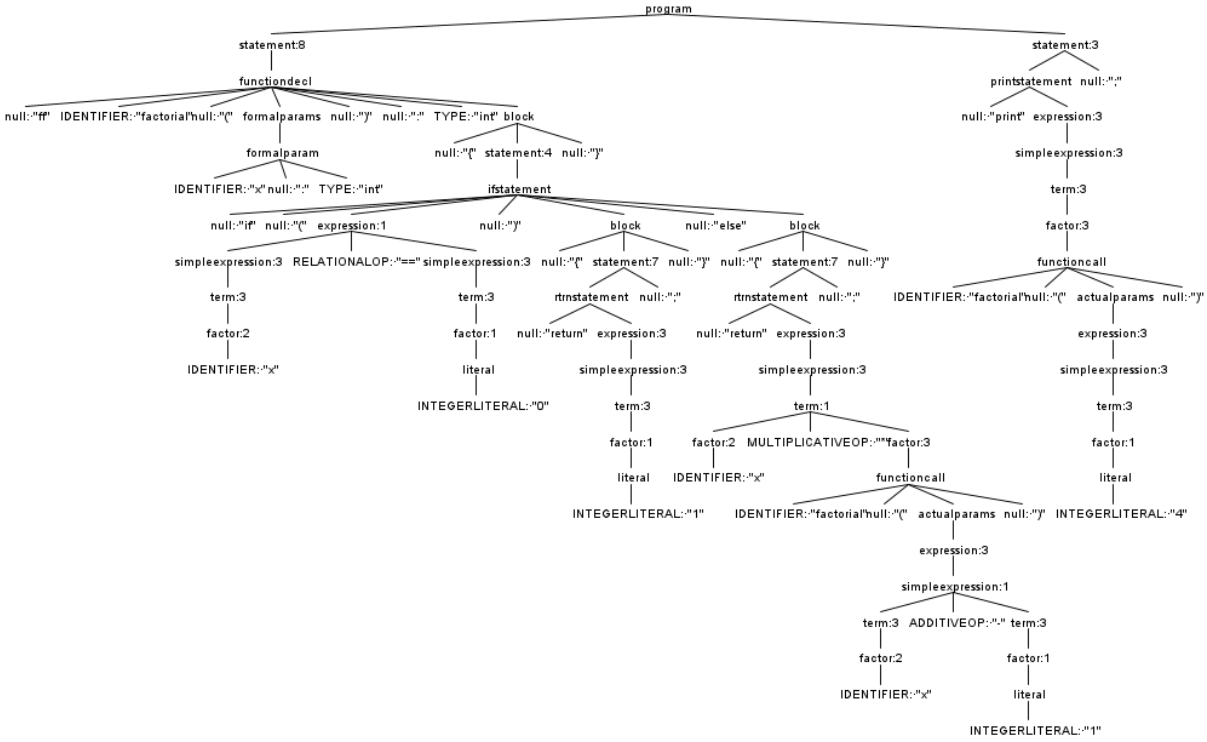


By comparing the above parse tree with the one shown previously, it can be concluded that the lexer rules devised to skip comments, were successful since the same parse tree was generated.

3.3.3 Parse Tree for recursive program

Using a recursive program, rules like those of function declaration and function call could be tested for correctness within the parse tree. Using the below input, the if and return statements are also tested for validity within the parse tree generated, as shown:

```
1  ff factorial(x:int) :int{
2      if( x == 0){
3          return 1;
4      } else {
5          return x * factorial(x-1);
6      }
7  }
8
9  print factorial(4); //24
```

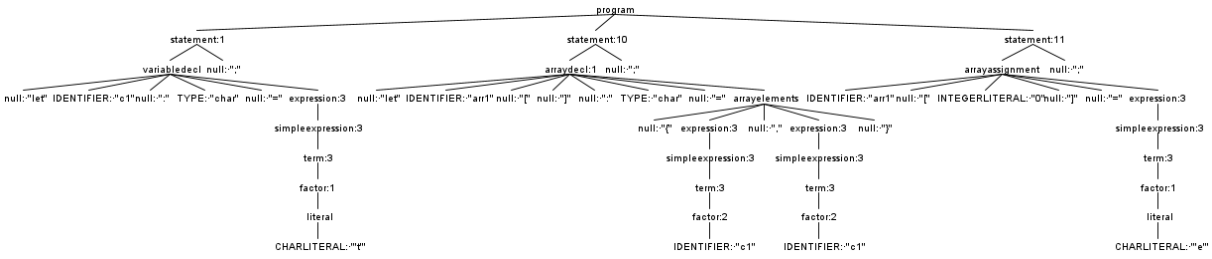


As can be seen, by increasing the input program by a few lines, the parse tree scales quite a bit to address the new statements. As shown, the root node correctly determines the presence of two global statements which are those of *functiondecl* and *printstatement*. On further investigation of the tree, it could be deduced that it is indeed correct, having the correct actual parameters for function calls, correct details for function declaration and correct handling of the *ifstatement* and both its blocks (if block and else block).

3.3.4 Parse Tree for SmallLangV2 program

The final parse tree to be tested for correctness is one making use of the SmallLangV2 rules for supporting character literals and arrays. The below input was used to generate the parse tree shown:

```
1  let c1:char = 't';
2
3  let arr1[]:char = {c1, c1};
4  arr1[0] = 'e';
```



As shown, the input makes use of a variable declaration of type char, an array declaration with an elements list and an array assignment. Other statements making use of rules such as the array call rule could have been done. This however was disregarded for such example to keep the parse tree at a smaller scale. As shown, by observing the variable declaration, the parse tree correctly contains the TYPE "char" and the expression value as a CHARLITERAL with value " 't' ". It is important to note that such character values will be stripped off their single quotes when the parse tree is converted to the AST. When observing the array declaration, no INTEGERLITERAL exists for the size of the array. This means that for no errors to be raised, an elements list had to be assigned to the newly declared array for the size to be determined from such list, in the future. If this wouldn't be present, an error would be shown as follows:

```
1  let c1:char = 't';
2
3  let arr1[]:char;
4  arr1[0] = 'e';
```

Error: line 3:15 mismatched input ';' expecting '='

The *arrayelements* are also correctly identified as the *c1* IDENTIFIER expressions. Moving on to the last statement, the array assignment within the parse tree is also correct, consisting of the correct IDENTIFIER, INTEGERLITERAL index, and CHARLITERAL assigned to it.

4. Hybrid Parser

In order to bypass the use of the hand-crafted parser, the ANTLR generated parser was used to generate the parse tree for a specific input program. The tree structure however does not resemble that of the AST generated by the hand-crafted parser and thus, needs to be converted. The following sub-section details how this conversion process was undertaken, and sub-section 4.3 tests the converted AST by running it through the three visitors created in the first compiler assignment.

4.1 Conversion Process

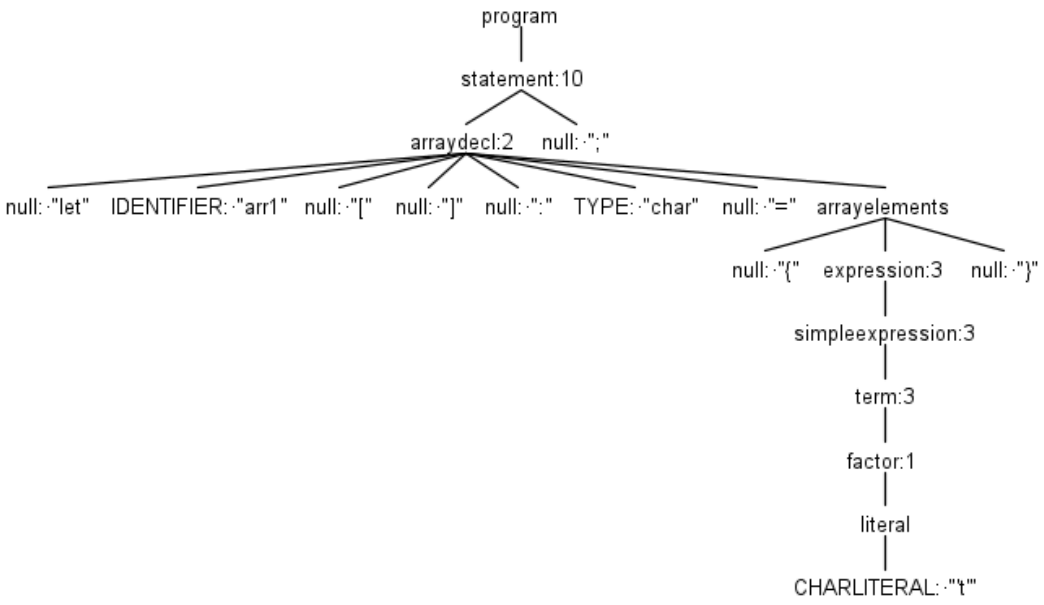
Since ANTLR has the option to generate visitor base classes and visitor interfaces for each grammar, it was decided that one such visitor would be used for converting the parse tree to an AST. This would be done, by visiting the nodes of the parse tree and were appropriate, return the AST node equivalent of the visited node. When generating/re-building the grammar, all ANTLR artefacts are overridden, and so, a custom visitor was implemented so that this static representation is never removed. Two visitors were implemented for each grammar, that is, *VisitorParseTreeToAst* and *VisitorParseV2TreeToAst*. The latter has all functions present in the former, along with the necessary functions for dealing with arrays and character values. For testing purposes, the *VisitorParseV2TreeToAst* class will be used since this has all features of the SmallLang language and more. Both classes were defined to implement the visitor interface associated with their grammar directory equivalents. *VisitorParseTreeToAst* implements the *SmallLangVisitor* whilst *VisitorParseV2TreeToAst* implements the *SmallLangV2Visitor*. By doing so, all functions present in the interface would have to be included in the visitor to be manipulated accordingly.

In ANTLR, the generated parse tree can be seen as a hierarchy of contexts, whereby the starting root node has a program context, and following that are the statement contexts that are children to the root node. These statements would have further contexts of their children and so on, until leaf nodes (lexer rules) are visited which have no child nodes. Such contexts will be widely used within the visitor classes for retrieving information from them and visiting other contexts if required. Also, the visitor design adopted doesn't include abstraction in function signatures, meaning that visit function identifiers were specified for the context they are dealing with. Example *visitBlock* being the function that handles the passed block context. In future implementations, this should be removed as it defeats the whole purpose of having a visitor, since it is required to specify each and every function call identifier, instead of just calling *visit* and passing the context to it. This implementation was used however to enforce a level of organisation and also since ANTLR's base interface included these pre-build functions with the detailed identifier names. All in all, the visitor implemented can be better described as a parse tree traverser where the tree is visited using depth-first search. This is because, when a node is accessed, its children will be accessed and if one of the children has its own child nodes, they will be accessed before continuing with the other children of the first node. To best explain how each AST node is generated, the next sub-section shows a dry run of one function within the *VisitorParseV2TreeToAst* visitor.

4.2 Dry Run Example

The following dry run was carried out to show one example of how a typical function within the implemented visitor classes, manipulates and gathers the necessary information from the passed context. Child contexts are also used to visit other functions and use the returned values to generate the valid AST node. The function used to carry out this dry run is the *visitArraydecl* function, since it shows the majority of checks undertaken by most visit functions. The following input program is used, and the parse tree generated is shown as well:

```
1    let arr1[]:char = {'t'};
```



Function: ASTArrayDecl visitArraydecl(SmallLangV2Parser.ArraydeclContext ctx)

1. On function entry, the variable *identifier* is initialised with a new *ASTIdentifier* node with the identifier value received by calling *ctx.IDENTIFIER().getText()*, which is "arr1". This call, uses the *arraydecl* context passed to the visit function, from the *statement* function (parent node), and retrieves the value stored in its IDENTFIER child node.

2. The function then proceeds to checks whether the context contains an INTERGERLITERAL child node (!= null) or whether it doesn't (== null). If the node exists, the function initialises the variable *arraySize* to the integer denoted by the INTERGERLITERAL node. In this case however, this process is skipped and the *arraySize* is left initialised to null.

3. The variable *type* is then initialised depending on the type the array was declared to be. Since the grammar was built to have a distinction between AUTO and TYPE, a TYPE node is first checked for whether it exists within the context. In this case it does, and so, the *type* variable is initialised with an *ASTType* node whose value is "char". In other cases, however, given that the array is declared with an auto type, the *type* variable would still be initialised with an *ASTType* node, but would have the value of "auto".

4. The *arrayElements* variable is then initialised by calling the *visitArrayelements* function and passing *ctx.arrayelements()* to it. This parameter refers to the context of the array elements child node, which is required for its visit function. Such function would first declare an empty list and then proceed to check if the context is null (doesn't exist) or not (exists). If it doesn't exist, an empty array list is returned, otherwise, an array list of expression nodes is returned with the expressions denoted in the input program. In this case an array list with one expression (CHARLITERAL) would be defined and an *ASTArrayElements* node is returned back to the function with the defined array list as its value. *arrayElements* would hence be initialised with this AST node.

5. Following this initialisation, the *arraySize* variable is then checked for whether it is equal to null and if so, is initialised to an *ASTIntegerLiteral* node with a value equal to the size of the elements list, which in this case is equal to 1.

6. Lastly, the function returns a new *ASTArrayDecl* node with a valid *identifier*, *arraySize*, *type* and *arrayElements*.

4.3 AST Testing

The following tests were carried out to determine if the AST produced from the parse tree conversion, produces correct results when given an input program. Within each test, the output from the *VisitorSemanticAnalysis* and *VisitorInterpreterExecution* passes are shown. The AST to XML generation output was not shown for any test, since these produced an overly large XML representation. By showing that the output from other passes is correct, the AST to XML visitor could be assumed to work just as well.

4.3.1 Testing large program

Test Input:

```
1  ff Square(x: float) : auto {
2      return x*x;
3  }
4
5  ff XGreaterThanY(x: float, y: float) : auto {
6      let ans:bool = true;
7      if(y > x) { ans = false; }
8      return ans;
9  }
10
11 ff AverageOfThree(x: float, y: float, z: float) : float {
12     let total:float = x + y + z;
13     return total/3.0;
14 }
15
16 let x:float = 2.4;
17 let z:int = 2;
18 let y:auto = Square(2.5);
19
20 print y; //6.25
21 print XGreaterThanY(x,2.3); //true
22 print XGreaterThanY(Square(1.5), y); //false
23 print AverageOfThree(x, y, 1.2); //3.28
```

Test Output:

```
Semantic Analysis Pass:
Success

Interpreter Execution Pass:
6.25
true
false
3.2833335
```

Output Description:

As shown, the AST generated from conversion produced a successful execution output where each outputted value is the same as those indicated within the inline comments on lines 20 to 23. In addition, no semantic errors were found, which was as expected.

4.3.2 Testing recursive program

Test Input:

```
ff factorial(x:int) :int{
    if( x == 0){
        return 1;
    } else {
        return x * factorial(x-1);
    }
}

print factorial(4); //24
```

Test Output:

```
Semantic Analysis Pass:
Success

Interpreter Execution Pass:
24
```

Output Description:

As shown, the AST generated from conversion produced a successful execution output whereby the answer '24' was printed out when executing the factorial of 4. In addition, no semantic errors were found, which was as expected.

4.3.3 Testing type-deduction program

Test Input:

```
ff XGreaterThanY(x: float, y: float) : auto {
    let ans:bool = true;
    if(y > x) {
        ans = false;
    }
    return ans;
}

let numX:auto = 2.5;
let numY:auto = 3.0;

print XGreaterThanY(numX, numY); //false
```

Test Output:

```
Semantic Analysis Pass:
Success

Interpreter Execution Pass:
false
```

Output Description:

As shown, the AST generated from conversion produced a successful execution output whereby the answer 'false' was printed out when executing the *XGreaterThanY* function to determine is *numX* is greater than *numY*. In addition, no semantic errors were found, which was as expected.

References

[1] Antlr.org. 2020. [online] Available at: https://www.antlr.org/papers/allstar-techreport.pdf [Accessed 23 May 2020].

[2] Antlr.org. 2020. *ANTLR*. [online] Available at: <https://www.antlr.org/> [Accessed 23 May 2020].

[3] Theantlrguy.atlassian.net. 2020. *Confluence*. [online] Available at: <https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687036/ANTLR+Cheat+Sheet > [Accessed 23 May 2020].