CPS2000 - Compiler Theory & Practise

Assignment Part 2

B.Sc Computer Science

Jacques Vella Critien - 97500L

Contents

1 Task1: Extending SmallLang

For the first task of this part of the assignment, we were required to extend SmallLang into SmallLangV2 by adding some other features. These features include adding support for the primitive type "char" and for arrays which hold a series of elements of the same type in contiguous memory. It was required to let array values uninitialised by default but an implementation for initialisation for values was also required. Moreover, formal parameters had to be changed in order to support both the "char" type and the arrays as types. In order to implement this, as can be seen below, EBNF rules had to be added and some were changed.

1.1 Solution

The rules below show the new and changed rules. The other rules which were present in SmallLang and not included in the below set, have not changed.

```
<ArraySizeIndex>        ::= '[' <Expression> ']'

<ArrayIdentifier>       ::= <Identifier> <ArraySizeIndex>

<ArrayValue>            ::= '{' <Expression> { ',' <Expression> } '}'

<VariableDecl> ::= <Identifier> ':' (<Type>|<Auto>) '=' <Expression>

<ArrayDecl>    ::=  <ArrayIdentifier>`:' <Type> ['=' <ArrayValue>]

<FormalParam>           ::= <Identifier> [ '[' ']' ] : <Type>

<AbstractIdentifier>    ::= <Identifier> | <ArrayIdentifier>

<Assignment>   ::= <AbstractIdentifier> `=' <Expression>

<Decl>                  ::= 'let' (<VariableDecl> | <ArrayDecl>)

<CharLiteral>           ::= '\'' <Letter> '\''

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

<Factor>                ::= <Literal>
                          | <AbstractIdentifier>
                          | <FunctionCall>
                          | <SubExpression>
                          | <Unary>

<Statement>             ::= <Decl> ';'
                         |    <Assignment ';'
                         |    <PrintStatement> ';'
                         |    <IfStatement>
                         |    <ForStatement>
                         |    <WhileStatement>
                         |    <RtrnStatement> ';'
                         |    <FunctionDecl>
                         |    <Block>
```

1.1.1 <ArraySizeIndex>

This rule represents the size of the array in the case of a declaration while it represents the index to assign in an assignment. It consists of an expression in the middle of square brackets. The expression would then be checked by the semantic analyser to make sure that it is of type int.

1.1.2 <ArrayIdentifier>

This rule represents an array identifier and it consists of an identifier followed by the above rule, which represents the size or the index.

1.1.3 <ArrayValue>

This rule represents the value to set to the array on declaration. This consists of an expression or more inside curly brackets.

1.1.4 <VariableDecl>

This rule represents a variable declaration for an array. I updated it by removing the 'let' from the start and starting with an ¡Identifier¿ node before a semi colon and a type which can also be auto. Finally, it remains the same by expecting an equal sign and an expression

1.1.5 <ArrayDecl>

This rule represents the declaration for an array. It starts with an <ArrayIdentifier> node explained above before a semi colon and a type. As can be seen in the figure above, an equals sign and an <ArrayValue> node are optional because arrays can be initialised or uninitailised in declarations.

1.1.6 <Decl>

Similarly, this new rule just represents either a variable declaration or an array declaration node by first expecting a let and then, either type of declaration.

1.1.7 <Assignment>

This rule is an updated version of the <Assignment> rule from part 1 of this assignment. As can be seen, this rule now accepts an ASTAbstractIdentifier which includes both ASTArrayIdentifier and ASTIdentifier rather than just ASTIdentifier.

1.1.8 <FormalParam>

This rule is an updated version of the <FormalParam> rule from part 1 of this assignment. As can be seen, optional empty square brackets are possible after the identifier which indicates an array as a formal parameter. Despite it is listed as an identifier, the actual code in the parser looks for a trailing '[' and if it is found an ASTArrayIdentifier node is returned and not an ASTIdentifier.

1.1.9 <AbstractIdentifier>

This new rule just represents either a normal identifier or an array identifier rule.

1.1.10 <CharLiteral>

This new rule was added to represent a character literal and it consists of a <Letter> rule in between two apostrophes.

1.1.11 <Literal>

This rule represents a literal and was updated to be able to also represent a <CharLiteral> rule.

1.1.12 <Factor>

This rule was updated to be able to represent an <AbstractIdentifier> rule instead of an <Identifier> rule to be able to also represent an <ArrayIdentifier> rule.

1.1.13 <Statement>

This rule was updated to be able to represent a <Decl> rule instead of an <VariableDecl> rule to be able to also represent an <ArrayDecl> rule.

2 Task1: SmallLangV2 Lexer and Parser

The second task required was to implement the necessary changes for the lexer and parser in order to process the input program containing the new features, namely the character literal and arrays. In order to perform this, I started off by extending the DFA (Deterministic Finite Automaton) to be able to split the inputs into correct tokens. Moreover, as will be explained below, the three tables which are the “Classifier Table”, the “Type Token Table” and the “Transition table” were also changed. Finally, for the parser, new nodes were created and the Parser class was updated.

2.1 Deterministic Finite Automaton

The figure below shows the added items to the automaton in part 1 so that the new features can be applied. As can be easily seen, State S28 represents a ‘[’ token, State S29 represents the ‘]’ token and S32 represents a character literal token, whose lexeme is in the form of ‘<character>’. **Once again, it is important to note that for each state, any other character inserted which are not visible in the paths going out from that state ALL lead to an absorbing bad state. This is not included in the diagram just to keep the diagram clear.**

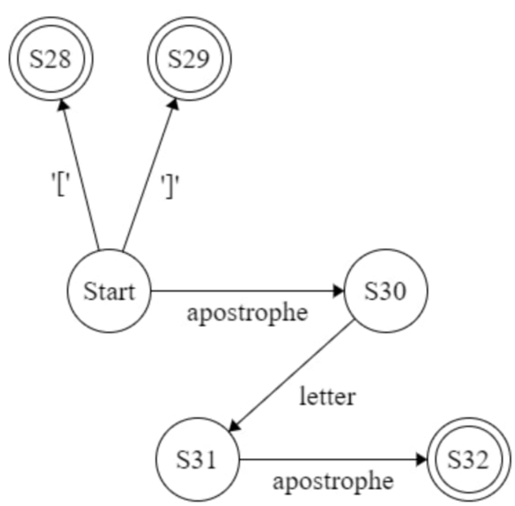

Figure 1: Deterministic finite automaton additions

2.2 Tables

2.2.1 Classifier Table

This table which relates the specific characters of input to the classifiers was updated in order to support the three new classifiers or categories. The new classifiers can be seen below and these were added to the the Classifier table created for part 1 of the assignment. The top row shows the character inputted and the bottom row shows the related classifier.

[	]	'
[	]	APOSTROPHE

Figure 2: Classifier Table additions

2.2.2 Type Token Table

This table which relates states to the classifiers was updated in order to support the five new states. The new states can be seen below and these were added to the the Type Token table created for part 1 of the assignment. The top row shows the state and the bottom row shows the related classifier.

S28	S29	S30	S31	S32
[	]	invalid	invalid	character

Figure 3: Type Token Table additions

2.2.3 Transition Table

This table which represents transitions from one state to another state when given a classifier, was updated in order to add the three new classifiers and the five new states. The transitions involving the new classifiers and states can be seen below marked in red. This was done to be able to distinguish them from previously created transitions for part 1 of the assignment. The columns represent the classifiers while the rows represent the states.

	DIGIT	DOT	LETTER	_	*	/	+	-	<	>	=	(	)	{	}	:	;	,	[	]	APOSTROPHE	NEWLINE	SPACE	EOF	OTHER
START	S1	BAD	S4	S4	S5	S6	S7	S8	S9	S10	S14	S16	S17	S18	S19	S20	S21	S22	S28	S29	S30	START	START	S27	BAD
S1	S1	S2	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S2	S3	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S3	S3	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S4	S4	BAD	S4	S4	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S5	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S6	BAD	BAD	BAD	BAD	S24	S23	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S7	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S8	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S9	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S11	S12	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S10	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S13	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S11	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S12	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S13	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S14	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S15	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S15	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S16	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S17	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S18	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S19	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S20	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S21	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S22	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S27	S23	S23	S23	S23	S23	BAD	BAD	BAD	START	S23	BAD	S23
S24	S24	S24	S24	S24	S26	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	BAD	BAD	BAD	S24	S24	BAD	S24
S25	S1	BAD	S4	S4	S5	S6	S7	S8	S9	S10	S14	S16	S17	S18	S19	S20	S21	S22	BAD	BAD	BAD	START	START	BAD	BAD
S26	S24	S24	S24	S24	S24	S25	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	BAD	BAD	BAD	S24	S24	BAD	S26
S27	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S28	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S29	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S30	BAD	BAD	S31	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S31	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S32	BAD	BAD	BAD	BAD
S32	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD

Figure 4: Transition Table additions

2.3 Lexer Solution

This section highlights and explains the difference and additions made in the code to support these new features in relation to the **lexer**.

2.3.1 TypeToken.java

This enum class which holds the different types of tokens was updated to include the following:

- SQUARE_OPEN
- SQUARE_CLOSE
- CHARACTER_LITERAL

2.3.2 Category.java

This enum class which holds the different types of categories or classifiers was updated to include these three new classifiers:

- SQUARE_OPEN
- SQUARE_CLOSE
- APOSTROPHE

2.3.3 State.java

This enum class which holds the different types of states was updated to include these 5 new states:

- S28
- S29
- S30
- S31
- S32

2.3.4 Keyword.java

This class which extends the Token class and in which all the keywords in the SmallLangV2 syntax are declared was updated and a new keyword to represent the char primitive was created and defined with the name CHAR.

2.3.5 Lexer.java

This class which contains all the methods needed from the parser to obtain the next token was updated to be able to handle the new features. Below contains all the list of methods that were changed and how:

1. **setTransitionTable()**: This function which populates the transition table hashmap was updated by adding the new transitions involved with the new classifiers and states. Basically, all the added transitions are the ones marked in red in the figure found in section 2.2.3
2. **setAcceptableStates()**: This function which populates the acceptable states hashmap was updated to include set states S28, S29 and S32 as acceptable states. These states can be confirmed as being acceptable and final from the automaton on section 2.1 and the Type Token table in section 2.2.2.

3. **charCat()**: This function which returns the category of a particular character was updated to support the three new tokens and categories which can be found in the classifier table in section 2.2.1

4. **nextToken()**: This method which is called by the parser to give out the next token was only changed in the last part, that is the result reporting by adding a clause to check if it is a character literal and if so, the apostrophes are removed from the lexeme. This can be seen from the code snippet below.

```
//if it is a character remove the apostrophes
else if(acceptableStates.get(state) == TypeToken.CHARACTER_LITERAL)
    return new Token(acceptableStates.get(state), lexeme.toString().substring(1,2));
```

Figure 5: Change in nextToken() method

2.4 Parser Solution

This section highlights and explains the difference and additions made in the code to support these new features in relation to the **parser**.

2.4.1 ASTAbstractIdentifier.java

This is a class which extends the **ASTExpression** interface. This is extended by the **ASTIdentifier** and **ASTArrayIdentifier** classes. This class has the following 2 members:

1. **name**: Its type is String and it is used to hold the variable name

2. **type**: Its of type Type (enumeration) and it is used to hold the type of the identifier.

In addition, this has getters for each member and a setter for the type to be used in case the identifier is of type auto so that it could be set to the expression's type as I will be explaining later.

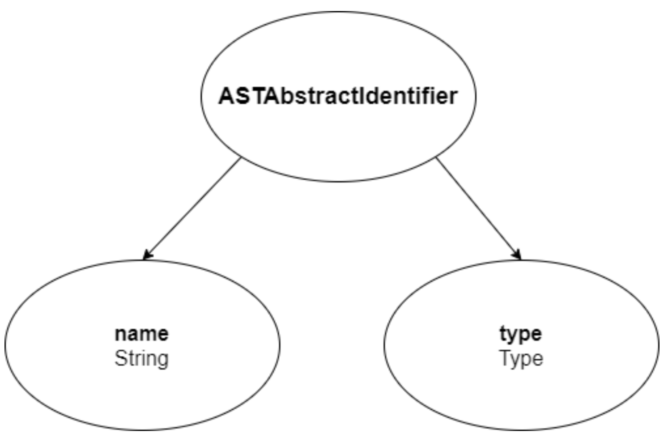

Figure 6: ASTAbstractIdentifier node

2.4.2 ASTIdentifier.java

This is a class which was created in part 1 of this assignment to represent an identifier. Now, it has been changed to extend the **ASTAbstractIdentifier** class and take up all of its member variables and methods which were explained in the above subsection highlighting the **ASTAbstractIdentifier** class.

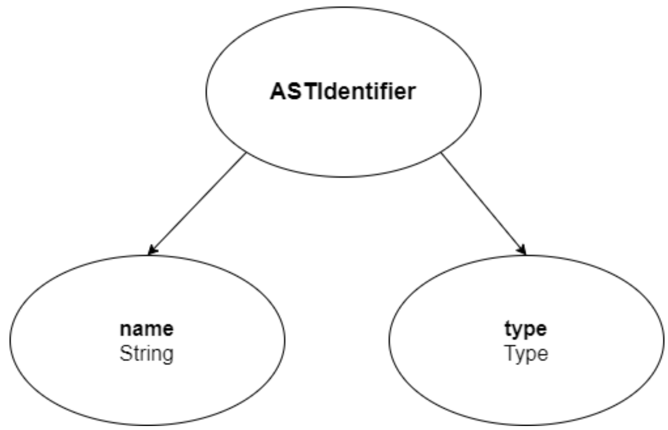

Figure 7: ASTIdentifier node

2.4.3 ASTArrayIdentifier.java

This is a class which extends the **ASTExpression** interface. This is extended by the **ASTIdentifier** and **ASTArrayIdentifier** classes. This class has the following 3 members:

1. **name**: Its type is String and it is used to hold the variable name
2. **sizeIndex**: Its of type ASTExpression and it is used to hold the size or index of the array identifier.
3. **type**: Its of type Type (enumeration) and it is used to hold the type of the identifier.

In addition, this has getters for each member, some of which are inherited from the ASTAbstractIdentifier class.

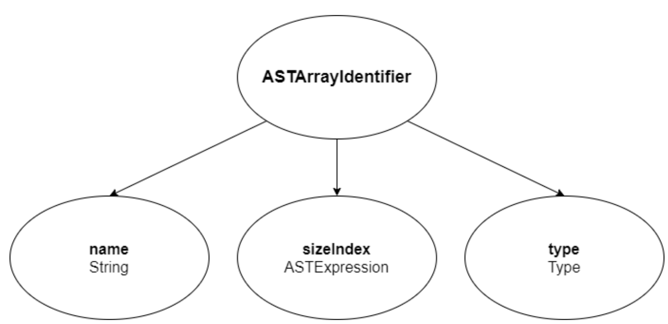

Figure 8: ASTArrayIdentifier node

2.4.4 ASTArrayValue.java

This is a class which extends the **ASTNode** interface. This was created to represent the value used to initialise an array, This class also has a member variable named values which is an arraylist of expressions of the type **ASTExpression**. In addition, this class also consists of constructors to create an object of this type,

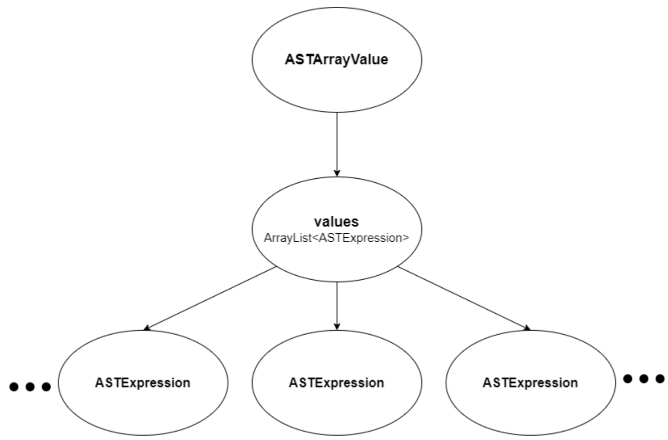

Figure 9: ASTArrayValue node

2.4.5 ASTDecl.java

This is a class which extends the **ASTStatement** interface. This is extended by the **ASTVariableDecl** and **ASTArrayDecl** classes.

2.4.6 ASTVariableDecl.java

This is a class represents a variable declaration and was declared in part 1 of this assignment. The only change to this class was to make it extend the **ASTDecl** class.

2.4.7 ASTArrayDecl.java

This class was added to represent an array declaration. It extends the newly ASTDecl class and contains the following two member variables:

1. **values**: Its type is ASTArrayValue and it is used to hold the array values to be declared. This can be left empty if the array is declared but not initialised.
2. **identifier**: Its of type ASTArrayIdentifier and it is used to identifier of the newly created array

In addition, this also contains a constructor to create a new instance of this class.

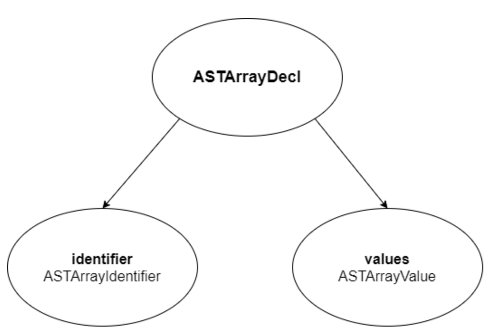

Figure 10: ASTArrayDecl node

2.4.8 ASTAssignment.java

This is a class which was created in part 1 of this assignment to represent an assignment. Now, it has been changed so that its member variable which represents the **identifier** is changed to be of the type of ASTAbstractIdentifier instead of ASTIdentifier so that it would support both an ASTIdentifier and an ASTArrayIdentifier.

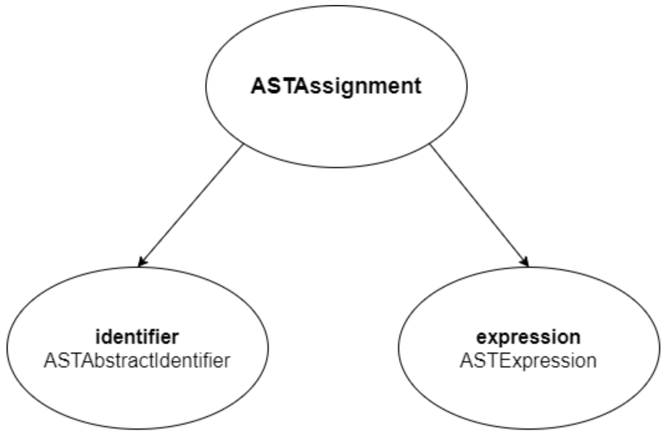

Figure 11: ASTAssignment node

2.4.9 ASTFormalParam.java

This is a class which was created in part 1 of this assignment to represent a formal parameter. Now, it has been changed so that its member variable which represents the **identifier** is changed to be of the type of ASTAbstractIdentifier instead of ASTIdentifier so that it would support both an ASTIdentifier and an ASTArrayIdentifier

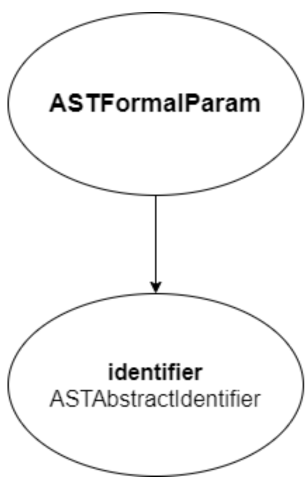

Figure 12: ASTFormalParam node

2.4.10 ASTCharacterLiteral

This class was added to the other AST classes. This class extends the **ASTExpression** class and represents a char literal. This class contains only one member variable name **value** and a constructor.

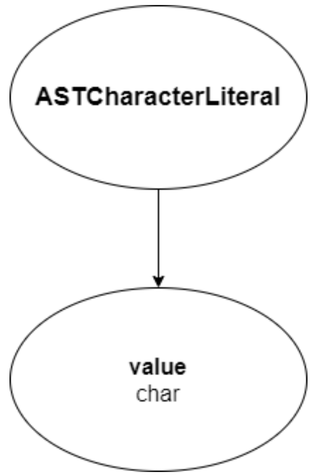

Figure 13: ASTCharacterLiteral node

2.4.11 Parser.java

This is the class in which one can find methods for parsing the input program. Below contains all the list of of new methods and methods that were changed and how:

1. **literal()**: This function checks the current token and returns an AST node according to its type. In order to cater for the new feature to allow for character literals, a new switch case was added to this method to return an ASTCharacterLiteral node if the type of the token is CHARATCER_LITERAL.

2. **arraySizeIndex()**: This function is used to parse an array size or index. The EBNF rule for this is defined as **‘[‘ <EXPRESSION>‘]’**, hence, this function first absorbs a token of the type SQUARE_OPEN, then gets the expression and finally, absorbs a SQUARE_CLOSE token before returning the expression obtained.

3. **arrayIdentifier(ASTIdentifier)**: This function is used to parse an array identifier. The EBNF rule for this is defined as **<IDENTIFIER><ARRAYSIZEINDEX>**. Moreover, this function accepts an ASTIdentifier as a parameter. The function first gets the expression by calling arraySizeIndex() and then returns a new ASTArrayIdentifier node by passing the identifier passed as a parameter and the expression obtained.

4. **factor()**: This function is used to parse a factor. This was created in part 1 of the assignment but was changed in order to cater for character literals and arrays functionalities. This was done by adding a a case to the switch where the type of the token is checked. If the type is of type CHARACTER_LITERAL, the function literal() is called. To cater for array identifiers, the case for when the token type is an identifier was modified by checking if the token after the identifier is of type SQUARE_OPEN, because if so, it must mean that it is an array identifier. In fact, if this is the case the function arrayIdentifier() explained above, is called.

5. **arrayValue()**: This function was created to parse an array value which may be used when declaring an array. The EBNF rules defines this as **’{’ <EXPRESSION>{ ’,’ <EXPRESSION>} ’}’**. In order to follow this rule, this function starts by defining an arraylist of expressions of type ASTExpression to hold values in it. After this, a {’ token is absorbed and the first expression is obtained and added to the array list. After this, there is a loop which goes on until as long as there are more commas to be parsed. Inside this loop, a new expression is obtained and added to the list of values.

```
//while there is more commas (more expressions)
while(this.currentToken.getType() == TypeToken.COMMA)
{
    //absorb the comma
    absorb(TypeToken.COMMA);

    //get the next value and add it to the list
    ASTExpression newExpression = expression();
    values.add(newExpression);
}
```

Figure 14: While loop in method

After all the values are obtained by not finding any more commas, a ‘}’ token is absorbed and a new ASTArrayValue node is returned with the values found.

6. **assignment()**: This function is used to parse an assignment and it was changed in order to also support an array identifier to be assigned. This was done by creating two new node explained above named **ASTAbstractIdentifier** and **ASTArrayIdentifier**. Rather than only calling identifier() and using only an ASTIdentifier, now this function makes use of an ASTAbstractIdentifier object to hold the identifier so that both an ASTIdentifier and an ASTArrayIdentifier could be held. Moroever, this method changed to first call identifier() and store this in the variable holding the identifier and then checking if there is a ‘[‘ token after the identifier which indicates that it is an array identifier. If this is the case, the identifier is passed as a parameter to the call made to the arrayIdentifier() method to obtain the arrayIdentifier. Finally, a new ASTAssignment node is created, this time with the new ASTAbstractIdentifier.

7. **declaration()**: Since a new ASTDecl class was created in order to represent both a variable declaration and an array declaration, this function was created as an entry point to the functions variableDeclaration(ASTIdentifier) and arrayDeclaration(ASTIdentifier). In fact, this is confirmed by the newly created EBNF rule to define a declaration which is defined as **<VARIABLEDECL>— <ARRAYDECL>**. This function first checks if there is a LET token and if not an empty ASTDecl node is returned, as may happen in a for loop with no declaration. Otherwise, the LET token is absorbed, the identifier is obtained by calling identifier() and then it is checked if the next token is of type SQUARE_OPEN. If it is, it means that it is an array declaration hence a call to the arrayDeclaration() method is done with the identifier passed as parameter. Otherwise, a call to the variableDeclaration() method is done with the identifier passed as a parameter.

8. **variableDeclaration(ASTIdentifier)**: This function is used to parse a variable declaration and it was changed by adding an the identifier as a parameter rather than obtaining it in the function itself. This is done since the function declaration(), explained above, will be called first and then this function is called from it.

9. **arrayDeclaration(ASTIdentifier)**: This function was created to parse an array declaration and it takes in an identifier as a parameter. The EBNF rule for an array declaration is defined as **’let’ <IDENTIFIER><ARRAYINDEX>’:’ <TYPE>**[**’=’ <ARRAYVALUE>**]. The LET token and the identifier are obtained by the declaration() function explained above which initiates this function. Then to continue following the rule, this function gets the array’s size by calling arraySizeIndex(), then absorbs the COLON token, gets and sets the type to the identifier and absorbs the TYPE token. Then, it is checked if there is a value by checking if the next token is of type ‘=’. If there is, it is absorbed and the value is obtained by calling arrayValue(). Otherwise, the ASTArrayValue node is left empty. Finally, a new ASTArrayDeclaration node is returned with the identifier and the value nodes.

10. **formalParam()**: This function is used to parse a formal parameter and it was updated to match its update EBNF rule which is defined as **(>IDENTIFIER>— >ARRAYIDENTIFIER>) ['[' ']'] ':' >TYPE>**. This function now starts by getting the identifier and storing it into an ASTAbstractIdentifier object since it can be both a normal identifier and an array identifier. Then, it is checked if the next token is of type SQUARE_OPEN, because if it is, it means that the formal parameter is an array. If so, '[' and ']' tokens are absorbed. After that, as used to happen before, a COLON token is absorbed and the type is obtained and set to the identifier. Finally, a new ASTFormalParam node is returned with the identifier of type ASTAbstractIdentifier.

11. **statement()**: This function is used to parse a statement and it was updated to be able to parse an array declaration. This was done by calling the newly created declaration() function instead of variableDeclaration() in the case of a LET token. Then, as explained above, the declaration() function would decide whether to call variableDeclaration() or arrayDeclaration() itself.

2.5 Unrequired Changes

The following contains explanation to simple changes made to the visitor classes for completion. It is important to note that these changes were not required in the assignment specification and were only done for completion of the visitor classes.

2.5.1 Visitor.java

This interface was changed to include all visit methods for newly created AST classes.

2.5.2 VisitorXMLGenerator.java

In this class the visit methods for **ASTCharacterLiteral, ASTArrayValue, ASTArrayDecl, ASTDecl** and **ASTArrayIdentifier** were added for completion. The visit method for the ASTDecl class was implemented to just cater for when the declaration is empty for the case of for loops with no declaration as this was changed from the parser to return an empty ASTDecl class rather than an empty ASTVariableDecl class. Moreover, the visit method for an ASTArrayIdentifier class was also implemented since it was easy and similar to the one of a variableDeclaration. It is important to note that some other changes had to be performed since some of the ASTClasses member variables' types changed and hence some variables used in this class had to be updated, such as ASTIdentifer to ASTAbstractIdentifier and ASTVariableDeclaration to ASTDecl.

2.5.3 VisitorSemanticAnalysis.java

In this class the visit methods for **ASTCharacterLiteral, ASTArrayValue, ASTArrayDecl, ASTDecl** and **ASTArrayIdentifier** were added for completion but were not implemented. It is important to note that some other changes had to be performed since some of the ASTClasses member variables' types changed and hence some variables' types used in this class had to be updated, such as ASTIdentifer to ASTAbstractIdentifier and ASTVariableDeclaration to ASTDecl.

2.5.4 VisitorInterpreter.java

Similarly, In this class the visit methods for **ASTCharacterLiteral, ASTArrayValue, ASTArrayDecl, ASTDecl** and **ASTArrayIdentifier** were added for completion but were not implemented. Moreover, some other changes had to be performed since some of the ASTClasses member variables' types changed and hence some variables' types used in this class had to be updated, such as ASTIdentifer to ASTAbstractIdentifier and ASTVariableDeclaration to ASTDecl.

2.6 Testing

In order to test my updated versions of the lexer and parser so that they can cater for the SmallLangV2 syntax, I continued to add to the tests I had prepared for part 1 of the assignment as can be seen below.

2.6.1 Lexer testing

In order to test my changes to the lexer, I added 3 new tests to test a character declaration, an array declaration and an array assignment. Moreover, I changed my input file for the function call test by adding an extra array formal parameter. These tests are performed by preparing a list of expected tokens and then this is compared to the list of tokens returned by the lexer and the test passes or fails whether the two lists would be exactly equal to each other.

2.6.2 Parser testing

On the other hand to test the changes to the parser, I had to create the following classes.

1. **VisitorChecker.java** : This is a visitor class which extends the Visitor interface. This was created in order to test each node in the resultant AST tree produced by the parser. This class contains these two member variables:

   - **sum**: This is a variable of type int which is used to hold the running sum.
   - **visitedIndexes**: This is an arraylist of type Integer which holds all the visited indexes which represents nodes.

   This works by having a visit method for each AST class and each node contains a unique index. Every time a node is visited, its index is added to the running sum and is inserted to the list containing the visited indexes. Obviously in each visitor method, the sub-nodes of that node are also visited.

2. **ParserTest.java** : This is a test class which tests new input program snippets which I created. These snippets were made sure to target all new target features. In fact, these include an array assignment, an correct array declaration, an incorrect array declaration, an array declaration with no initialisation, array as a formal parameter a character declaration and an array assignment. This works by first parsing the input program, then the VisitorChecker class mentioned above is used by making it visit all the nodes of the tree returned by the parser. Finally, the sum member variable and the size of the visited indexes arraylist of the VisitorChecker instance are asserted.

The input programs for the lexer and parser test can be found in the resources folder as can be seen i the image below.

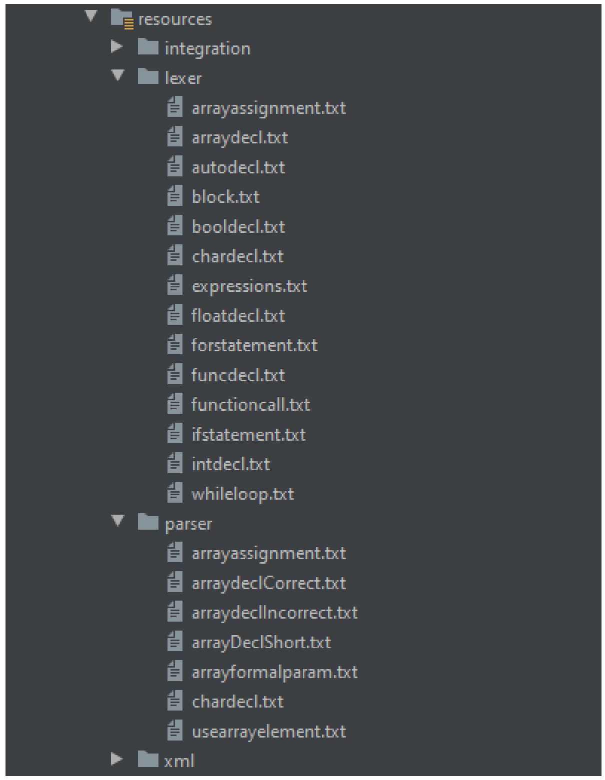

Figure 15: File Structure for input programs used for testing

3 Task3: Tool generated parsers

For this task, we were required to carry out research on ANTL, which is a tool used to generate a parser by giving it a grammar. Therefore, we had to create a grammar for both SmallLang and SmallLangV2. Finally, in order to be able to verify the correctness of the generated parser, we were required to check and compare with the AST generated by the hand-crafter parser explained above.

3.1 Solution

Since this implementation was done using Java and coded using the IntelliJ IDE, ANTLR's plugin[1] was used to help in implementing the grammar and seeing its results in the form of parse trees or hierarchies by using the GRUN tool.

3.1.1 SmallLang

After spending several hours researching about ANTLR grammars, I started implementing the grammar for SmallLang by creating a SmallLang.g4 file in the "src/main/antlr" directory. In order to create this grammar, I followed the EBNF rules found in the specification for part 1 of this assignment and tried to replicate them. The file contents can be found in the image below. It is important to note that multiplicative operators, additive operators and relational operators were not listed as lexer rules but were listed as parser rules, that is, starting with a lowercase letter because, for an unknown reason, the'and' and'or operators as for an unknown reason they were being ignored when multiplicate operands and additive operands were listed as lexer rules.

```
grammar SmallLang;

@header {...}

literal : BooleanLiteral | IntegerLiteral | FloatLiteral;
multiplicativeOp : TIMES | DIVIDE | AND;
additiveOp : PLUS | MINUS | OR;
relationalOp : LT | GT | EQUAL | NOT_EQUAL | LTE | GTE;
actualParams :  expression (COMMA expression)*;
functionCall : Identifier BRACKET_OPEN actualParams? BRACKET_CLOSE;
subExpression : BRACKET_OPEN expression BRACKET_CLOSE;
unary : (MINUS | NOT) expression;
factor : literal | Identifier | functionCall | subExpression | unary;
term : factor (multiplicativeOp factor)*;
simpleExpression : term (additiveOp term)*;
expression : simpleExpression (relationalOp simpleExpression)*;
assignment : Identifier EQUAL_SIGN expression;
variableDecl : LET Identifier COLON (Type | Auto) EQUAL_SIGN expression;
printStatement : PRINT expression;
rtrnStatement : RETURN expression;
ifStatement : IF BRACKET_OPEN expression BRACKET_CLOSE block (ELSE block)?;
forStatement : FOR BRACKET_OPEN variableDecl? SEMI_COLON expression SEMI_COLON assignment? BRACKET_CLOSE block;
whileStatement : WHILE BRACKET_OPEN expression BRACKET_CLOSE block;
formalParam : Identifier COLON Type;
formalParams : formalParam (COMMA formalParam)*;
functionDecl : FF Identifier BRACKET_OPEN formalParams? BRACKET_CLOSE COLON (Type | Auto) block;
statement : variableDecl SEMI_COLON
          | assignment SEMI_COLON
          | printStatement SEMI_COLON
          | ifStatement
          | forStatement
          | whileStatement
          | rtrnStatement SEMI_COLON
          | functionDecl
          | block;
block : CURLY_OPEN statement* CURLY_CLOSE;
program : statement*;

fragment DIGIT : [0-9];
fragment LETTER : [A-Za-z];
LET : 'let';
NOT : 'not';
MINUS : '-';
EQUAL_SIGN : '=';
COLON : ':';
SEMI_COLON : ';';
BRACKET_OPEN : '(';
BRACKET_CLOSE : ')';
CURLY_OPEN : '{';
CURLY_CLOSE : '}';
COMMA : ',';
IF: 'if';
ELSE: 'else';
FOR : 'for';
FF : 'ff';
PRINT : 'print';
RETURN : 'return';
WHILE : 'while';
Type : 'float' | 'int' | 'bool';
Auto : 'auto';
AND : 'and';
OR : 'or';
TIMES : '*';
DIVIDE : '/';
PLUS : '+';
LT : '<';
GT : '>';
EQUAL : '==';
NOT_EQUAL : '<>';
LTE : '<=';
GTE: '>=';
BooleanLiteral : 'true' | 'false';
IntegerLiteral : DIGIT+;
FloatLiteral : DIGIT+ '.' DIGIT+;
Identifier : ('_' | LETTER) ('_' | LETTER | DIGIT)*;
WS : [ \r\t\n]+ -> skip ;
COMMENT : '//' .*? [\n] -> skip;
MULTI_LINE_COMMENT : '/*' .*? '*/' -> skip ; // .*?
```

Figure 16: Grammar for SmallLang

3.1.2 SmallLangV2

After implementing the grammar for SmallLang, constructing the grammar for SmallLangV2 was not difficult. Basically, all the changes in the EBNF rules explained in task 1 of this assignment, were made to the grammar. This included changes to the **literal, factor, assignment, formalParam** and **statement** rule. Moreover, new rules were added, namely, **arrayIndex, arrayDecl, arrayIdentifier, abstractIdentifier, arrayValue, declaration** and **CharLiteral**

```
grammar SmallLangV2;

/*
Done to add package on top of generated files
*/
@header {...}

literal : BooleanLiteral | IntegerLiteral | FloatLiteral | CharLiteral;
multiplicativeOp : TIMES | DIVIDE | AND;
additiveOp : PLUS | MINUS | OR;
relationalOp : LT | GT | EQUAL | NOT_EQUAL | LTE | GTE;
actualParams :  expression (COMMA expression)*;
functionCall : Identifier BRACKET_OPEN actualParams? BRACKET_CLOSE;
subExpression : BRACKET_OPEN expression BRACKET_CLOSE;
unary : (MINUS | NOT) expression;
factor : literal | abstractIdentifier | functionCall | subExpression | unary;
term : factor (multiplicativeOp factor)*;
simpleExpression : term (additiveOp term)*;
expression : simpleExpression (relationalOp simpleExpression)*;
assignment : abstractIdentifier EQUAL_SIGN expression;
arrayIndex : SQUARE_OPEN expression SQUARE_CLOSE;
variableDecl : Identifier COLON (Type | Auto) EQUAL_SIGN expression;
arrayDecl : arrayIdentifier COLON Type EQUAL_SIGN arrayValue;
arrayIdentifier : Identifier arrayIndex;
abstractIdentifier: Identifier | arrayIdentifier;
arrayValue : CURLY_OPEN expression (COMMA expression)* CURLY_CLOSE;
declaration : 'let' (variableDecl | arrayDecl);
printStatement : PRINT expression;
rtrnStatement : RETURN expression;
ifStatement : IF BRACKET_OPEN expression BRACKET_CLOSE block (ELSE block)?;
forStatement : FOR BRACKET_OPEN variableDecl? SEMI_COLON expression SEMI_COLON assignment? BRACKET_CLOSE block;
whileStatement : WHILE BRACKET_OPEN expression BRACKET_CLOSE block;
formalParam : Identifier (SQUARE_OPEN SQUARE_CLOSE)? COLON Type;
formalParams : formalParam (COMMA formalParam)*;
functionDecl : FF Identifier BRACKET_OPEN formalParams? BRACKET_CLOSE COLON (Type | Auto) block;
statement : declaration SEMI_COLON
          | assignment SEMI_COLON
          | printStatement SEMI_COLON
          | ifStatement
          | forStatement
          | whileStatement
          | rtrnStatement SEMI_COLON
          | functionDecl
          | block;
block : CURLY_OPEN statement* CURLY_CLOSE;
program : statement*;

fragment DIGIT : [0-9];
fragment LETTER : [A-Za-z];
LET : 'let';
NOT : 'not';
MINUS : '-';
EQUAL_SIGN : '=';
COLON : ':';
SEMI_COLON : ';';
BRACKET_OPEN : '(';
BRACKET_CLOSE : ')';
CURLY_OPEN : '{';
CURLY_CLOSE : '}';
SQUARE_OPEN : '[';
SQUARE_CLOSE : ']';
COMMA : ',';
IF: 'if';
ELSE: 'else';
FOR : 'for';
FF : 'ff';
PRINT : 'print';
RETURN : 'return';
WHILE : 'while';
Type : 'float' | 'int' | 'bool'| 'char';
Auto : 'auto';
AND : 'and';
OR : 'or';
TIMES : '*';
DIVIDE : '/';
PLUS : '+';
LT : '<';
GT : '>';
EQUAL : '==';
NOT_EQUAL : '<>';
LTE : '<=';
GTE: '>=';
BooleanLiteral : 'true' | 'false';
IntegerLiteral : DIGIT+;
FloatLiteral : DIGIT+ '.' DIGIT+;
CharLiteral : '\'' LETTER '\'';
Identifier : ('_' | LETTER) ('_' | LETTER | DIGIT)*;
WS : [ \r\t\n]+ -> skip ;
COMMENT : '//' .*? [\n] -> skip;
MULTI_LINE_COMMENT : '/*' .*? '*/' -> skip ; // .*?
```

Figure 17: Grammar for SmallLang

3.2 Checking correctness

The files used can be found in the directory "src/main/java./resources/lexer". In order to better compare these, both were translated to hierarchical viewing mode. The diagram on the left shows the tree generated by the hand-crafter parser while the diagram on the right shows the output by GRUN. As can be seen below, there were not many difference apart from the hand-crafted parser's output tree being more simplified and without tokens.

3.2.1 SmallLang

In order to check the correctness of the grammar used for SmallLang, small programs consisting of different features the language supports were created and then, the AST tree generated by the hand-crafted parser and the parse tree generated by GRUN were compared. It is important to add that for this validating the GRUN output for SmallLang, the AST classes were generated using the hand-crafter parser from part 1 of this assignment since the hand-crafter parser in this part of the assignment was changed to cater for SmallLangV2.

1. **Variable Declaration**: The file used for this program is named **intdecl.txt** and it highlights a variable declaration of type int. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary.In fact, the int and identifier tokens from the GRUN output are held into a single ASTIdentifier node in my implementation. As for the expression, my implementation only contains 1 node while GRUN shows all steps taken to finally arrive at the integer literal.

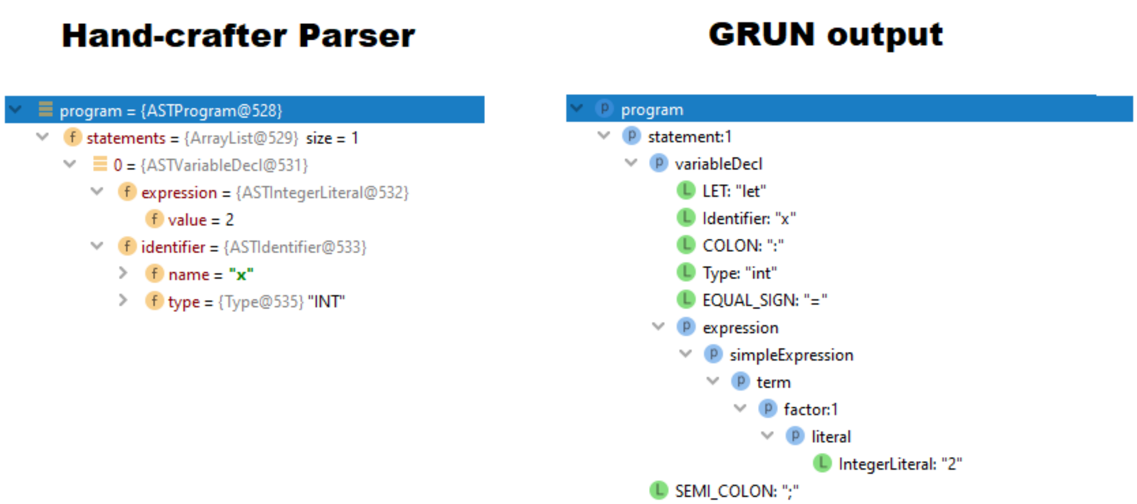

Figure 18: Comparison for a variable declaration

2. **Assignment**: The file used for this program is named **assignment.txt** and it highlights an assignment. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, the identifier is held into a single ASTIdentifier node in my implementation. As for the expression, my implementation only contains 1 node while GRUN shows all steps taken to finally arrive at the integer literal.

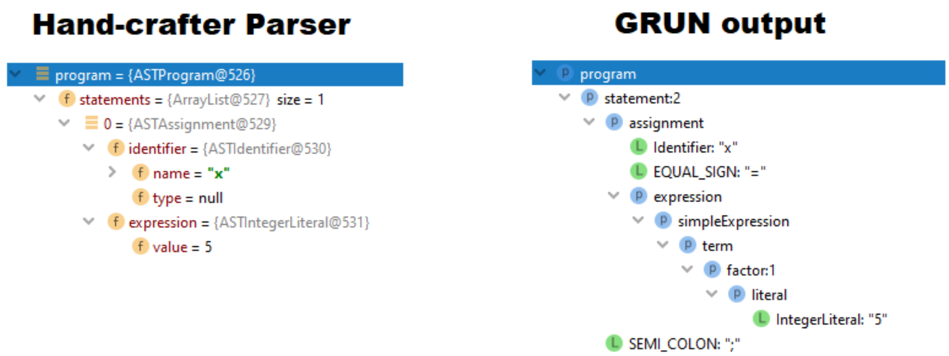

Figure 19: Comparison for an assignment

3. **Function Declaration**: The file used for this program is named **funcdeclv1.txt** and it highlights a function declaration. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, the function identifier and the return type are held into a single ASTIdentifier node in my implementation and the formal parameters are very similar but the ':' token is discarded. As for the block, my implementation does not hold then the curly brackets tokens and the expression is summarised into an ASTBinExpression node.

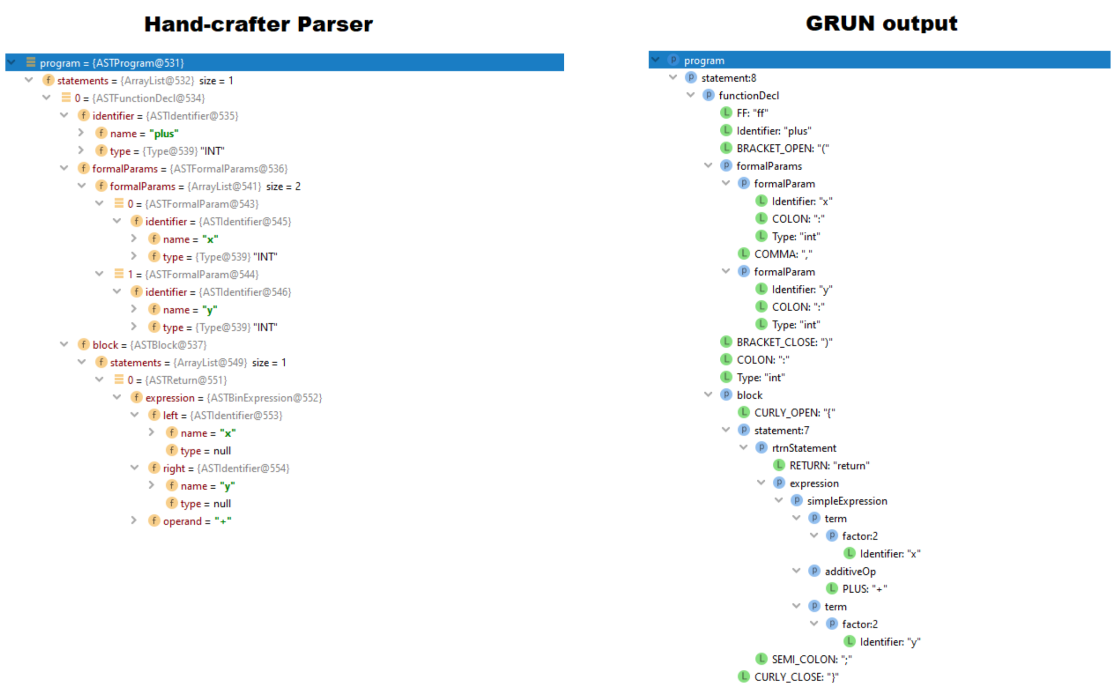

Figure 20: Comparison for a function declaration

4. **Function Call**: The file used for this program is named **functioncall.txt** and it highlights a

function call. For this example, we will be only looking at the part of the function call as the variable declaration differences were covered above. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, the function identifier is held into a single ASTIdentifier node in my implementation and the actual parameters in my implementation only consists of an array containing the expressions, in this case integer literals while GRUN shows all steps taken to finally arrive at the integer literals.

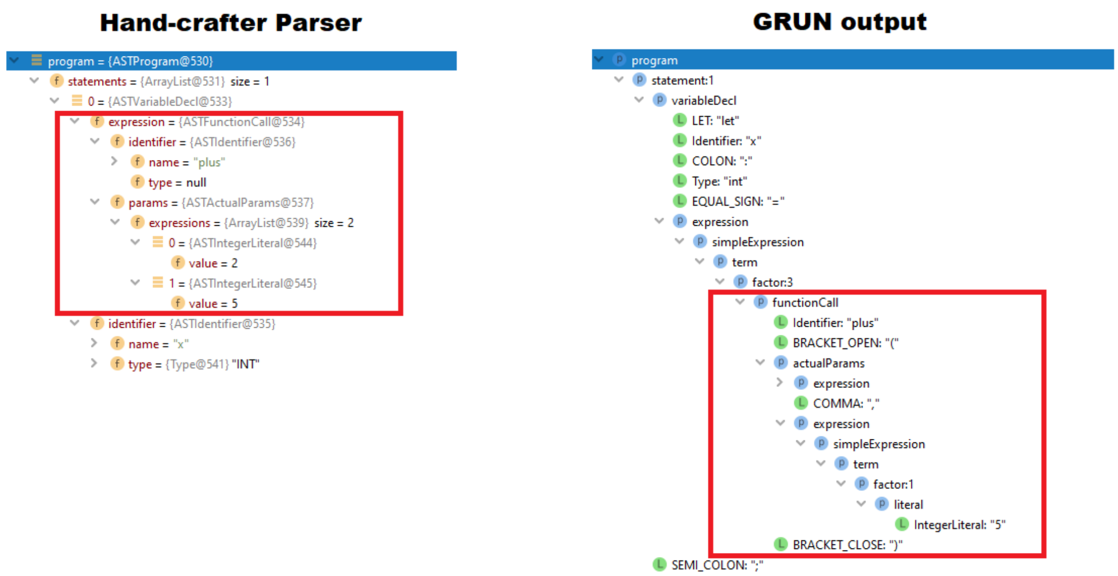

Figure 21: Comparison for a function call

5. **Block**: The file used for this program is named **block.txt** and it highlights a block. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, everything is the same apart that the GRUN output shows that it keeps the curly bracket tokens while my implementation does not.

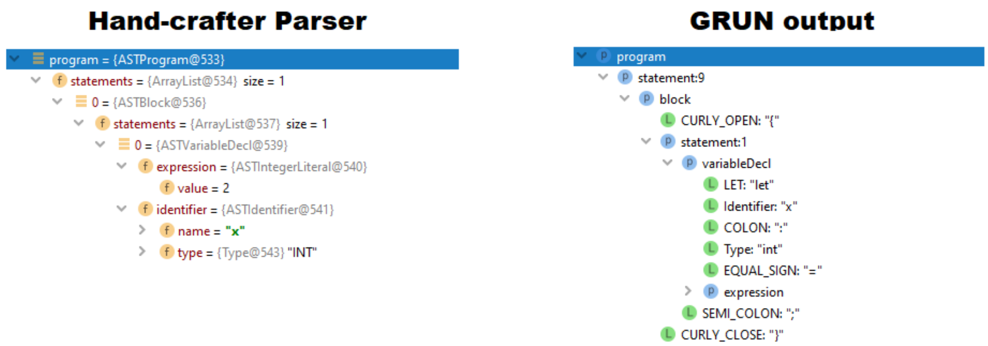

Figure 22: Comparison for a block

6. **If Statement**: The file used for this program is named **ifstatement.txt** and it highlights an if statement. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, everything is the same apart from the removal of the bracket, if and else tokens in my implementation. Obviously, the other differences in blocks and expressions are the same as the ones discussed above.

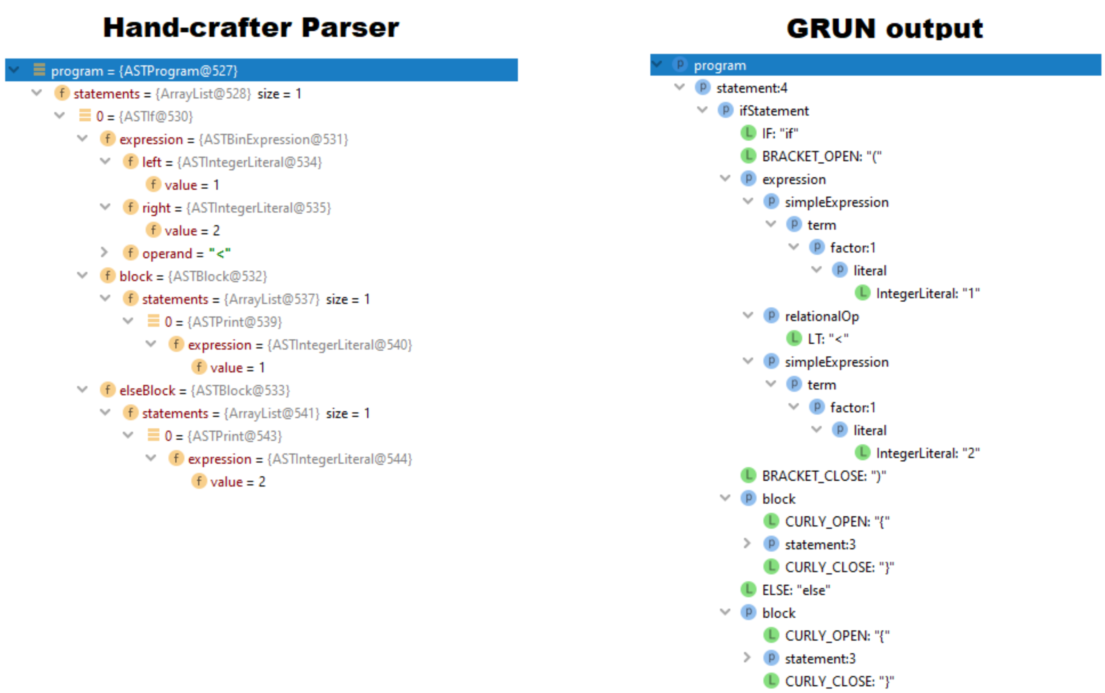

Figure 23: Comparison for an if statement

7. **While Loop**: The file used for this program is named **whileloop.txt** and it highlights a while loop. As can be seen in the comparison below, both trees are the conceptually same, however,

the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, everything is the same apart from the removal of the brackets and while tokens in my implementation. Obviously, the other differences in blocks and expressions are the same as the ones discussed above.

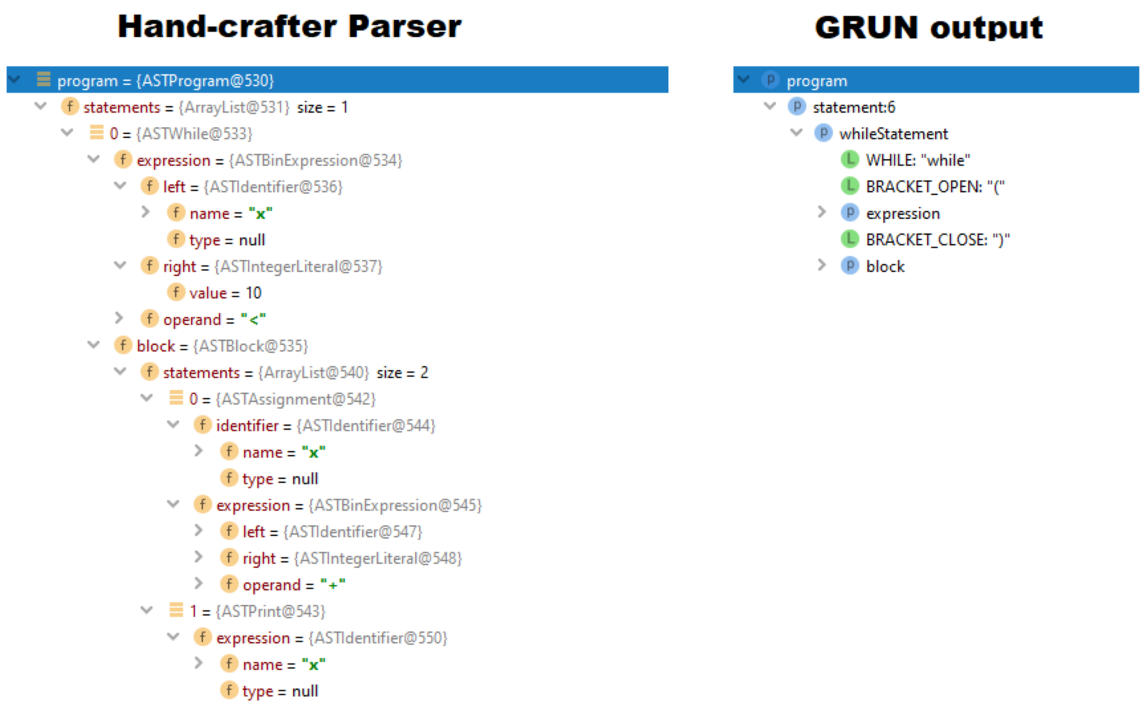

Figure 24: Comparison for a while loop

8. **For Loop**: The file used for this program is named **forstatement.txt** and it highlights a for loop. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The only difference is that my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, everything is the same apart that in my implementation the brackets, for and semi colons tokens after the declaration and expression are removed. Obviously, the other differences in blocks and expressions are the same as the ones discussed above.

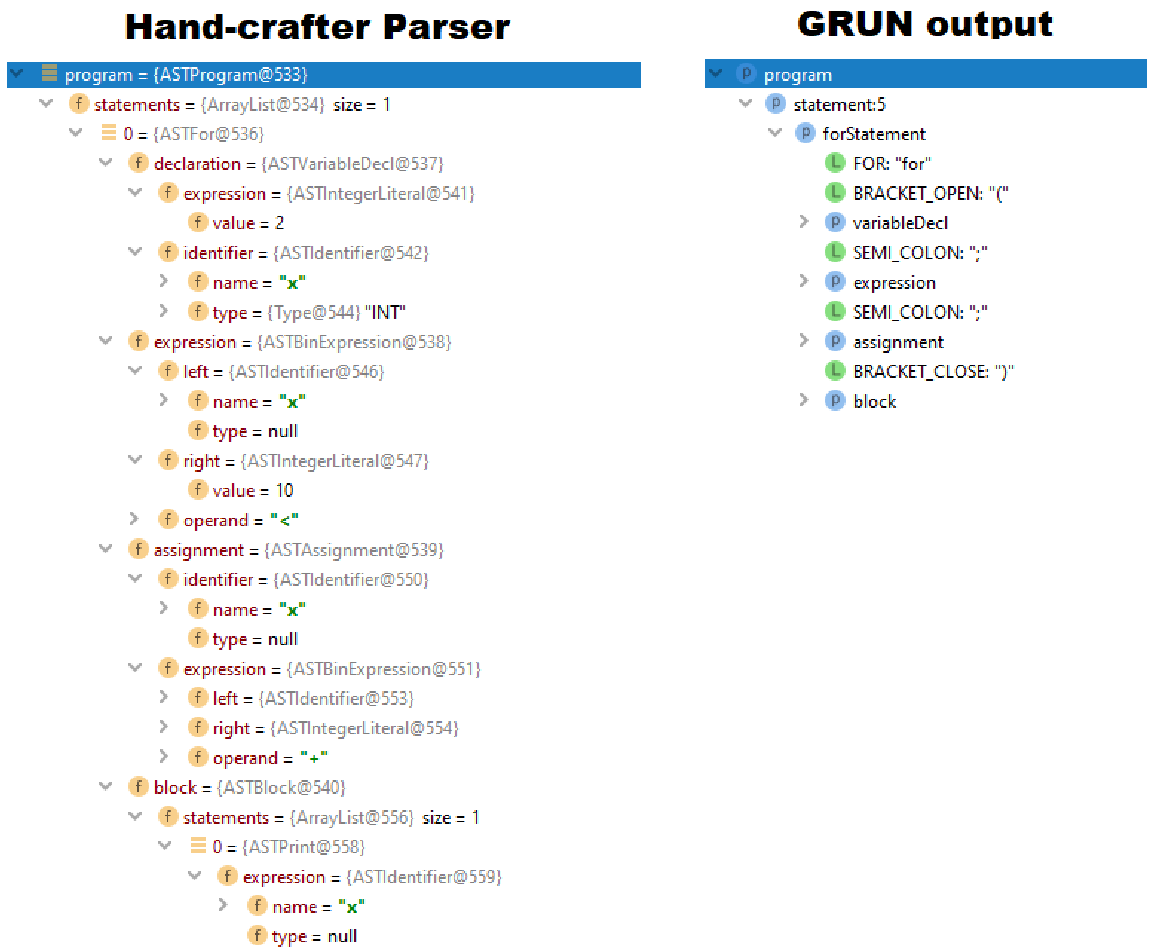

Figure 25: Comparison for a for loop

3.2.2 SmallLangV2

In order to check the correctness of the grammar used for SmallLangV2, small programs consisting of different snippets containing the new features of the language were created and then, the AST tree generated by the hand-crafted parser and the parse tree generated by GRUN were compared.

1. **Character Declaration**: The file used for this program is named **chardecl.txt** and it highlights a character declaration of type int. Despite being very similar, there are still notable differences between the two generated ASTS. The tree generated by GRUN shows all the tokens in that rule since it is a parse tree and it also includes the declaration while my hand-crafted parser identifies these tokens and only keeps what is necessary. In fact, the char and identifier tokens from the GRUN output are held into a single ASTIdentifier node in my implementation. As for the expression, my implementation only contains 1 node while GRUN shows all steps taken to finally arrive at the character literal. It is also good to note that my class does not contain a declaration node but only a variable declaration node because my ASTVariableDecl extends the ASTDecl class.

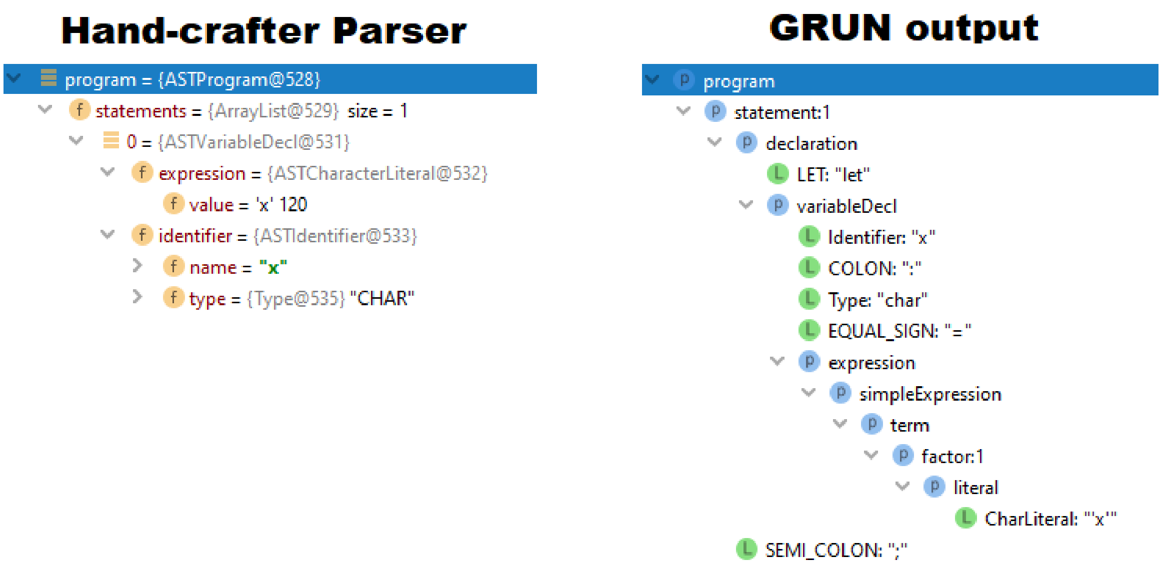

Figure 26: Comparison for a character declaration

2. **Array Declaration**: The file used for this program is named **arraydecl.txt** and it highlights an array declaration of type int. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. The differences between the two are that my implementation of the parser discarts the let and brackets tokens. This can also be noticed in the array size index where the square brackets are eliminated and in the array value where the curly brackets are eliminated. Apart from that, the structure is similar.

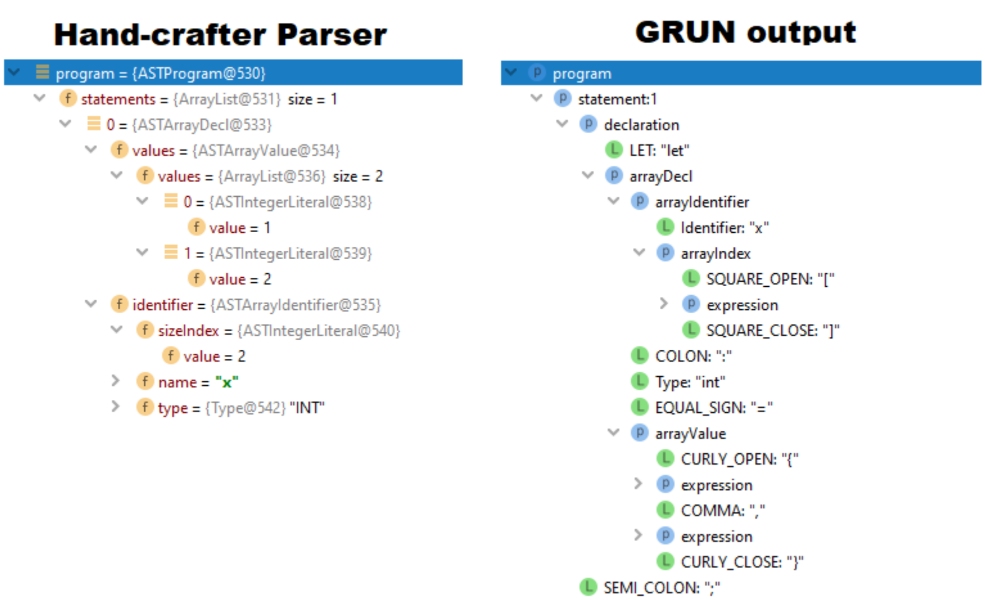

Figure 27: Comparison for an array declaration

3. **Array Assignment**: The file used for this program is named **arrayassignment.txt** and it highlights an array assignment of type int. As can be seen in the comparison below, both trees are the conceptually same, however, the tree generated by GRUN shows all the tokens in that rule since it is a parse tree. Apart from the extra tokens, my implementation of the parser does not produce an ASTArrayIdentifier inside an ASTAbstractIdentifier but only produces an ASTArrayIdentifier since an ASTArrayIdentifier extends an ASTAbstractIdentifier, as if they are the same.

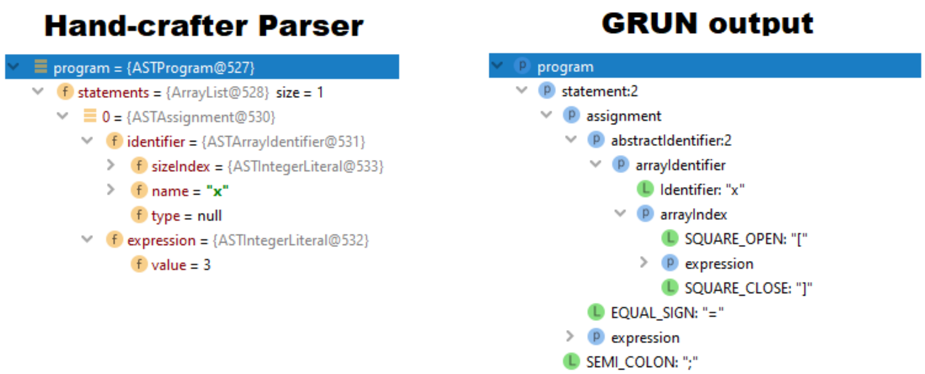

Figure 28: Comparison for an array assignment

4. **Array formal parameter in function declaration**: The file used for this program is named **funcdecl.txt** and it highlights a function declaration but with a formal parameter which is an array. As can be seen in the comparison below, both trees are the conceptually same, however, the formal parameter which is an array is of the type ASTArrayIdentifier rather than of the type ASTIdentifier.

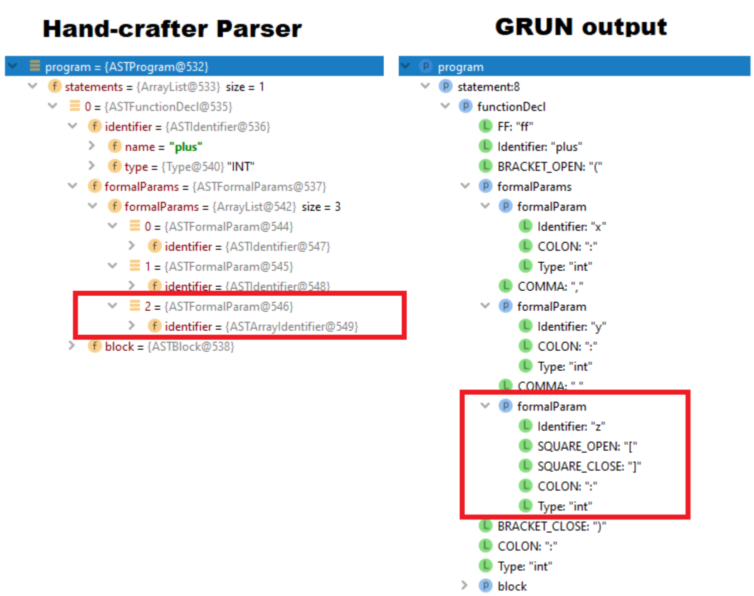

Figure 29: Comparison for an array formal parameter

References

[1] "Antlr v4 grammar plugin." https://plugins.jetbrains.com/plugin/7358-antlr-v4-grammar-plugin. Accessed on 07-05-2020.

[2] "Java with antlr." https://www.baeldung.com/java-antlr. Accessed on 04-05-2020.

[3] "Antlr 4 documentation." https://github.com/antlr/antlr4/blob/master/doc/index.md. Accessed on 04-05-2020.

[4] "Antlr beginner tutorial 2: Integrating antlr in java project." https://www.youtube.com/watch?v=itajbtWKPGQ. Accessed on 05-05-2020.

[5] "Antrl v4 on intellij, part 2 - visitors." https://www.youtube.com/watch?v=dPWWcH5uM0g&t=305s. Accessed on 06-05-2020.

[6] "Antlr4 - how do i get the token type as the token text in antlr?." https://stackoverflow.com/questions/38106771/antlr4-how-do-i-get-the-token-type-as-the-token-text-in-antlr. Accessed on 08-05-2020.

[7] "Handling errors in antlr4." https://stackoverflow.com/questions/18132078/handling-errors-in-antlr4/18137301#18137301. Accessed on 08-05-2020.