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.

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.**

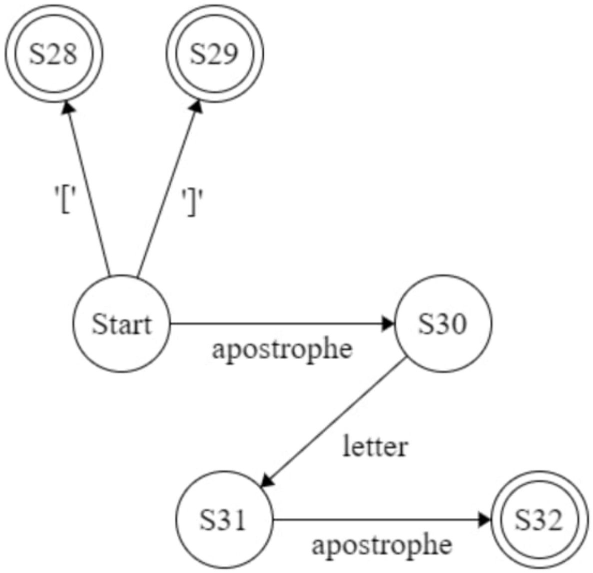
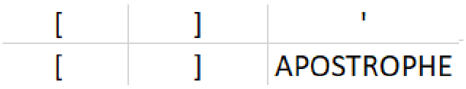


Figure 1: Deterministic finite automaton additions

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.

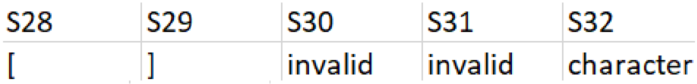


[	]	'
[	]	APOSTROPHE

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.

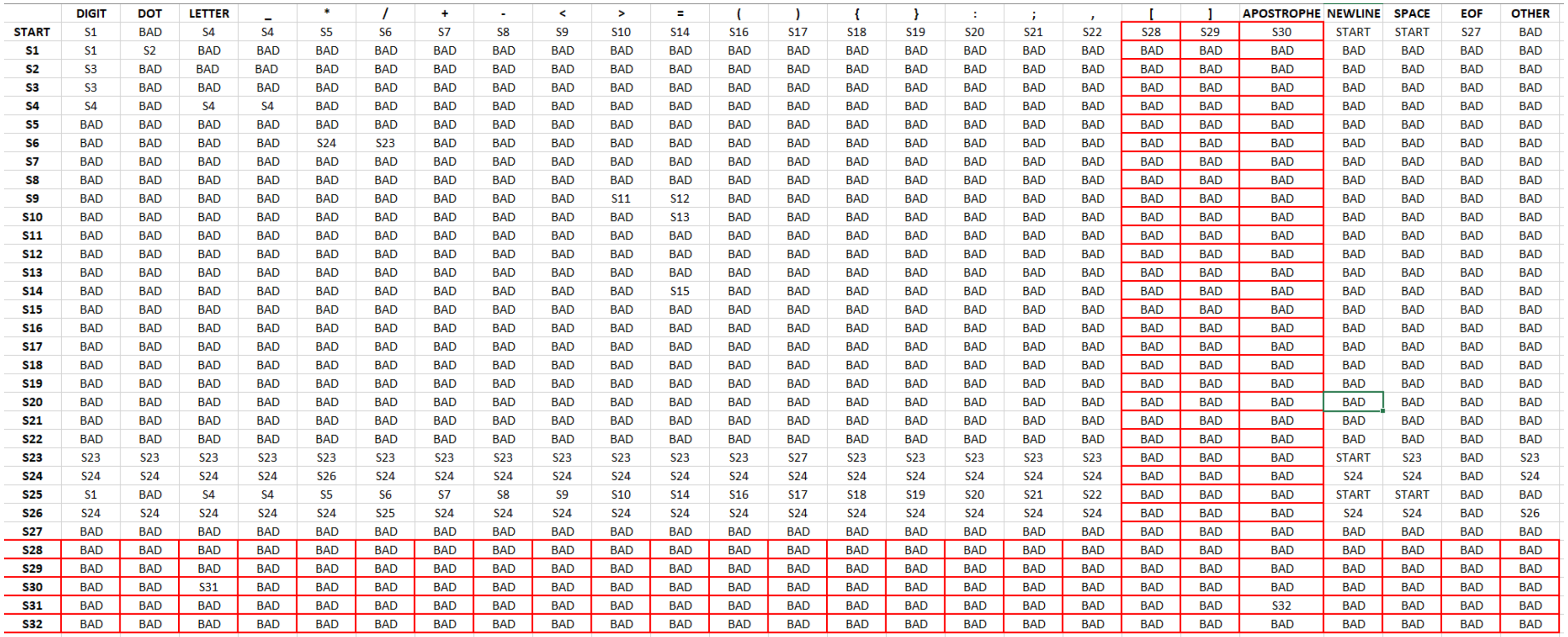


S28	S29	S30	S31	S32
[	]	invalid	invalid	character

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

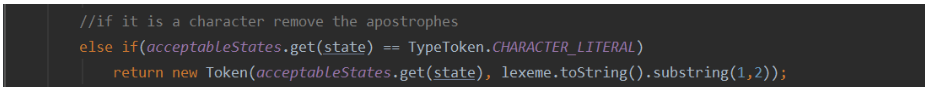
This class which extends the Token class and in which all the keywords in the SmallLangV2 syntax are declared was updated and a new keyword to represent the char primitive was created and defined with the name CHAR.

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.

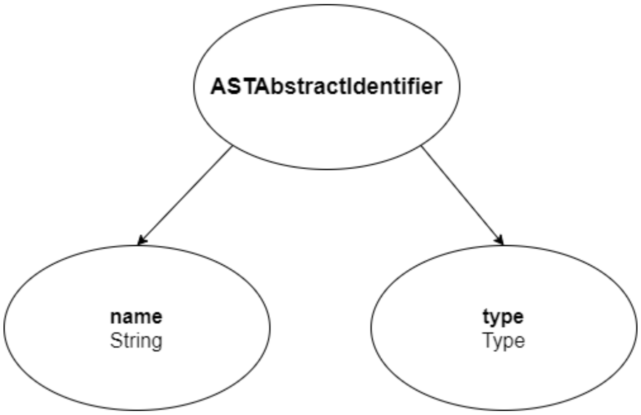


Figure 6: ASTAbstractIdentifier node

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.

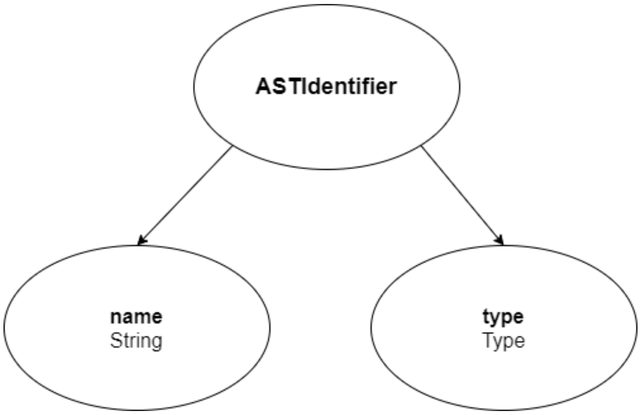


Figure 7: ASTIdentifier node

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.

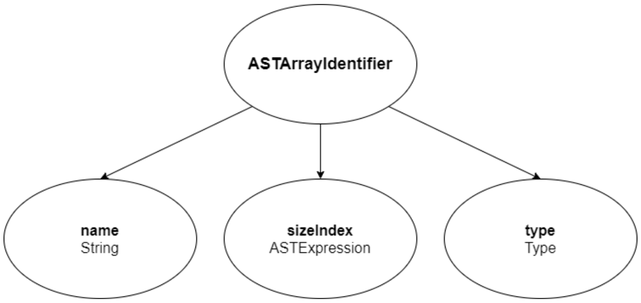


Figure 8: ASTArrayIdentifier node

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,

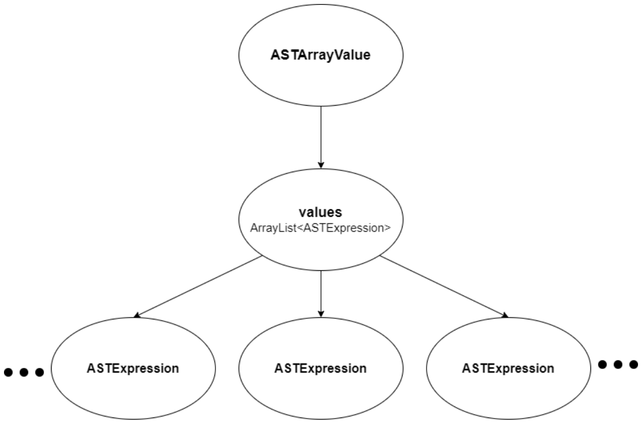


Figure 9: ASTArrayValue node

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.

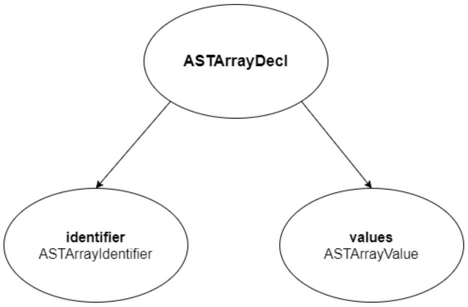


Figure 10: ASTArrayDecl node

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.

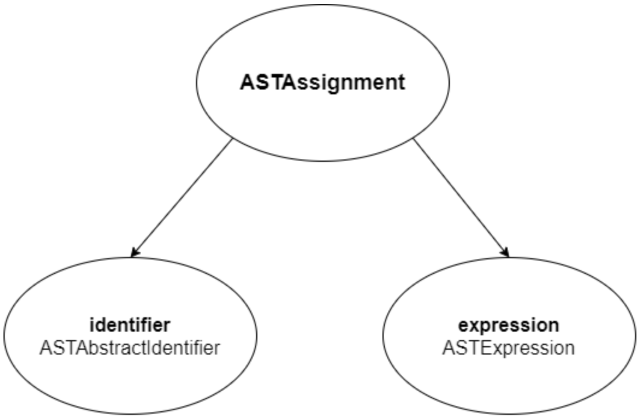


Figure 11: ASTAssignment node

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

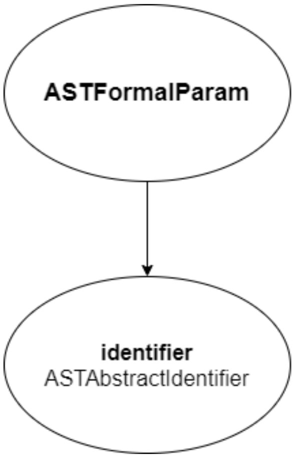


Figure 12: ASTFormalParam node

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.

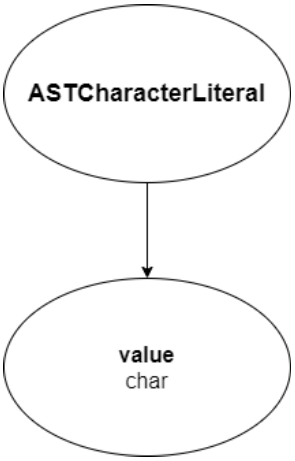


Figure 13: ASTCharacterLiteral node

References

[1] “Java with antlr.” https://www.baeldung.com/java-antlr. Accessed on 04-05-2020.

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

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

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

[5] “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.

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