Testing Grammars For Top-Down Parsers

A.M. Paracha and F. Franek
Dept. of Computing and Software
McMaster University, Hamilton, Ontario

ABSTRACT

According to the software engineering perspective, grammars can be viewed as *"Specifications for defining languages or compilers"*. They form the basics of languages and several important kinds of software rely on grammars; e.g. compilers and parsers, debuggers, code processing tools, software modification tools, and software analysis tools. Testing a grammar to make sure that it is correct and defines the language for which it is developed is also very important.

We implemented Purdom's algorithm to produce test data automatically for testing the MACS [1] grammar (an LL(1) grammar) and the parser. Two different implementations of Purdom's algorithm were carried out in this project, one in Java and the other in C++; test strings and other analysis data automatically generated by these implementations are given in the paper.

I. INTRODUCTION

During the past several years, complexity of compilers has grown much and so is the importance of testing them. [10, 12, 13] Compiler essentially is a software tool and hence its testing should fulfill all the software testing criteria. Testing is the process of finding errors in an application by executing it. It is one of the essential and most time consuming phases of software development. Hence a lot of effort is directed to fully automate this process, making it more reliable, repeatable, less time consuming, less boring and less expensive.

A compiler is a computer program that accepts a source program as input and produces either an object code, if the input is correct, or error messages, if it contains errors. Compilers are tested to establish some degree of correctness. The basic testing relies on ***Test Cases***. A test case for a compiler should have [2]:

1. A test case description.
2. A source program for which the output of the compiler under observation is verified.
3. An excepted output.

In this project, we were testing MACS compiler whose top-down parser is based on an LL (1) grammar, so our test data are program fragments correct with respect to the MACS grammar used. Test cases should cover all possible valid and invalid input conditions. One of the major problems in generating test cases is the completeness of coverage and the potentially unfeasible size of the test data.

When generating test data for compilers they should cover all the syntax and semantic rules of the language and generate errors in all possible contexts. If upon executing a test case, the output matches the excepted one (including the error messages generated), then the compiler passed the test. On the other hand, if the generated output and/or errors if applicable do not match, the compiler have errors and should be corrected. [14]

The rest of the paper is organized as follows. Section 2 gives details about grammars and how to test them. Section 3 presents issues regarding parsers and their testing, and section 4 presents Purdom's algorithm in detail. In section 5 we give details about our implementation and the results generated. Section 6 concludes the paper and gives remarks on future research directions.

II. TESTING GRAMMARS

Compilation is the process of transformations of the input program written in a source language to a program in the target language. Traditionally (since the advent of Algol 60 programming language), the source language is specified by means of a formal grammar. A grammar is the main input for the test case generation process. There are a wide variety of grammars, however two play an important role in programming language design and the compilation process, namely context free grammars (used to define the control constructs of the language), and regular grammars (used to define lexical terms).

A grammar defines both a language and provides a basis for deriving elements of that language, grammar is considered both as a program and a specification. [3]

A ***Context Free Grammar*** (CFG) is a set of (possibly) recursive rewriting rules (also called productions) used to generate strings of alphabet symbols in various patterns. There are four major components in a grammar **<*N*, *T*, *s*, *P*>**, where ***N*** is a finite set of so-called non-terminal symbols (non-terminals); ***T*** is a set of so-called terminal symbols

[1] MACS is a name for a programming language used in the forthcoming book of Franek on compilers. MACS is an acronym for McMaster Computer Science

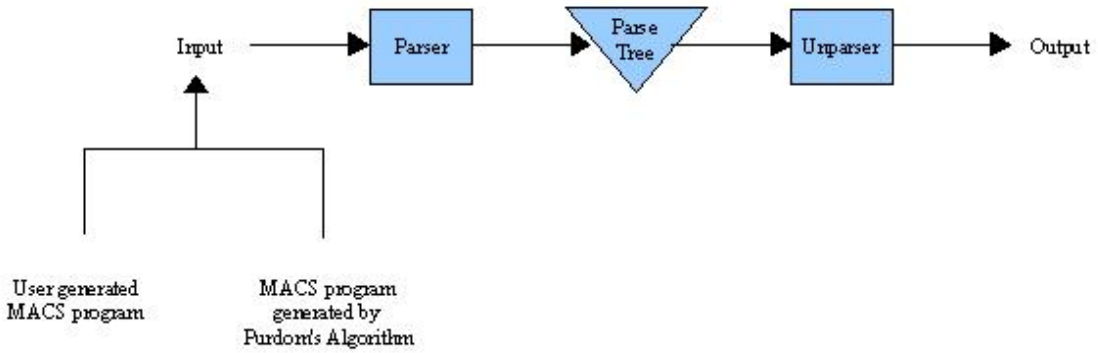

Fig. 3.1 Validation Testing

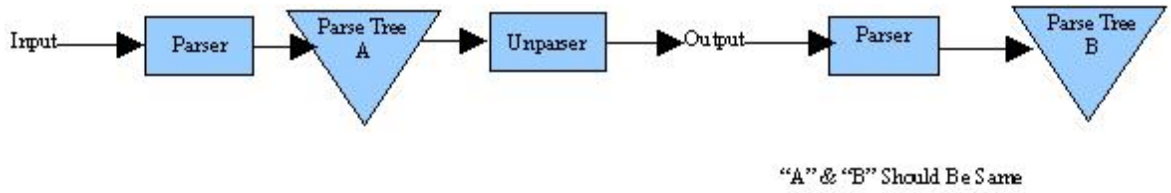

Fig. 3.2 Testing the Parser

(terminals or tokens), which does not intersect with ***N***; ***s*** is an initial symbol from ***N*** (starting non-terminal), and ***P*** is a finite subset of the set $N \times (N \cup T)^*$. Pairs from the set ***P*** are called ***grammar rules*** or ***productions***. A rule (***n***, ***m***) is usually written as $n \rightarrow m$. The set of all sequences derivable from the starting symbol *s* of the grammar G and containing only terminal symbols is called the ***language generated*** by the grammar G and is denoted as L(G). Such sequences are called ***language phrases*** or ***language sentences***. A context-free grammar only defines the syntax of a language; other aspects such as the semantics cannot be covered by it. Every symbol of the grammar defines some finite set of attributes, each production corresponds to a set of rules for evaluating these attributes. [7,9]

To derive a program (sentence) in a language using a grammar, we begin with the start symbol ***s*** and apply the appropriate productions to non-terminals, as rewriting rules until we are left with a sequence consisting entirely of terminals. This process generates a tree whose root is labeled by the start symbol, whose internal nodes are labeled by the non-terminals and whose leaves are labeled by the terminals. The children of any node in the tree correspond precisely to those symbols on the right-hand-side of the corresponding production rule. Such a tree is known as a parse tree, the process by which it is produced is known as parsing. [7,8, 9]
Testing a grammar for errors is very difficult. Grammars should be tested to verify that they define the language for which they were written, they should be tested for completeness, that is every non-terminal must have some derivation to be converted into a string of terminals, and that every rule is needed. Detection of errors in a grammar at an early stage is very important as the construction of compiler depends on it.

For compiler design, the grammar used is supposed to be unambiguous. An ambiguous grammar is the one that has more then one parse tree for some sentence. Ambiguity in grammar is to be removed because ambiguous grammars are problematic for parsing and can lead to unintended consequences for compilation, in essence detracting from the intended definition of the language. We have to check the grammar for ambiguity and if needed, to disambiguate it at an early stage.

There are no practical means to check the dynamic semantics of a language defined by a context-free grammar. This problem is addressed by program validation and verification. Here we are strictly concerned with the syntax analysis only.

For generating program test cases for a parser based on context free grammar, the grammar is in fact an ideal tool. The rules of the grammar if combined in a random or regular way can be used to generate sentences (or fragments of sentences) for that language. If a language is intended to be a computer programming language, then the sentences represent programs or their fragments written in the language.

III. TESTING PARSER

Parsing, or syntax analysis, is one of the activities performed by the front end of a compiler and hence finds use in many software engineering tools such, as automatic generation of documentation, coding tools such as class browsers, metrication tools, and tools for checking code styles. Automatic re-engineering and maintenance tools, as well as tools to support refactoring and reverse-engineering also typically require a parser at the front end.

Parsing is the process whereby a given program is checked against the grammar rules to determine (at least) whether or not it is syntactically correct.[4]

Software testing activities are organized on how the test data is generated. Based on this criterion, there are two basic types of testing:

Black Box Testing: The test cases are developed without considering the internal structure of the software under investigation, only the desired behavior is considered and tested for.

White Box Testing: The test cases are prepared with the full knowledge of the inner structure of the software to be tested. One of the goals of the white box testing is to ensure that every control flow path in the software is exercised and tested. Applying this approach to testing parsers leads to the requirement that every production rule of the grammar is used at least once. [8,14]

Test data for a parser is a program that uses all the production rules of the underlying grammar. Generating such programs manually is difficult and error prone (and seldom complete), so we need to have a method for automatic generation of test data for the parser.

As mentioned above, the grammar used for parsing and for generating the test data must be unambiguous to guarantee reasonable results.

While testing the parser, the input of the parser and the output of the unparser (see Figure 3.1) should both agree. Of course they can differ in indentation and other white space as these are generally ignored by compilers, redundant parentheses in expressions, and many other aspects. In other words, the input and the output must be similar, though not necessarily the same. If the input and the output are compared by humans, these differences can be easily determined and checked. However, for an automated comparison we need a more elaborate setting to determine if the input and the output are similar, see Figure 3.2

If the input and the output are not properly similar, we might have one of the following possibilities:

- The parser is incorrect.
- The unparser is incorrect.
- Both are incorrect.

IV. PURDOM'S ALGORITHM

Purdom in 1972 [1,15] proposed a method for testing compiler by automatically generating test programs on the basis of the grammar with the main objective of using each language rule at least once. According to him, a set of sentences using all the language rules has a good choice of exercising most of the compiler code or tables. As all the programming languages are context sensitive, this method only confirms the syntactical aspect and there is no guarantee that these programs will execute correctly.

Also Purdom's algorithm focus on verifying the compiler correctness not interested in checking the efficiency, performance and other aspects. Purdom's algorithm generates sentences that are correct with respect to the context-free grammar of the language, but may be inconsistent with respect of the contextual constraints such as variable declarations and use of identifiers. It only verifies the syntax analyzer of the compiler.

We used Purdom's algorithm to generate test cases for the parser of MACS compiler validation. It produces a set of sentences from the given context-free grammar of MACS, which are then used as test input for the parser.

How Purdom's Algorithm Work

Given a set of terminals, a set of non-terminals, a starting symbol, and a set of productions, it generates a shortest program in the language so that every rule has been used at least once. The emphasis is on speed and performance. The productions are used as rewrite rules for generating sequences of grammar symbols. The initial sequence consists of the start symbol. Repeatedly, a non-terminal in the sequence is replaced by the right-hand-side symbols of a rule that has the non-terminal as its left-hand-side. The process terminates when all symbols in the sequence are terminals. Since a MACS program is not a sequence of terminals, but a sequence of lexemes, the sequence of terminals is then translated (in a somehow arbitrary way) to a sequence of lexemes, i.e. a MACS program.

The output of Purdom's algorithm can be used only to check the syntax, but it generates short test suites rapidly and efficiently. One of the goals of Purdom's algorithm is to keep the length of the sentences as short as possible.

V. IMPLEMENTATION OF PURDOM'S ALGORITHM AND RESULTS

We have used Purdom's algorithm to test the parser for MACS grammar. It is an LL(1) grammar and thus can be used for top-down predictive parsing. Specification of the grammar is given in appendix A. The grammar has 77 terminals, 90 non-terminals, and 301 productions.

In our implementations, intermediate data structures needed to generate sentences are:

- Production number used to rewrite the symbol resulting in the shortest terminal string (SLEN).
- Production number used to introduce the symbol into the derivation of shortest string (DLEN)

In the implementations, the lengths of strings are calculated as:

For Terminals: Length=1 (We opted for this compromise, since the actual length of corresponding lexemes cannot often be determined. For instance, for the terminal COMMA we know that the length will always be 1 as there is only one lexeme for it ',' . On the other hand, the lexemes for ID can be of any length and so we would not know how to assign the lengths to the terminal ID)

For Non-terminals: Length=No. of terminals+ No. of steps for the derivation

We have implemented the algorithm in both C++ and Java. Both implementations are very closely related to the implementation and reformulation of Purdom's algorithm given by Malloy and Power in [5] and [6]. However, we ran into several problems with their implementation of phase III and so we had to choose our own approach to rectify it. In phase I and II, Purdom's algorithm generates several intermediate arrays, which are used to hold the intermediate results. Our implementation consists of three phases, each producing the following results:

A. Phase I (Shortest String Length)

In the first phase of Purdom's algorithm, we take the set of terminals, non-terminals, language rules and the start symbol "S", and for each symbol it calculates the following pieces of information which will be used later in our sentence generation procedure.

Input:

terms.asc	set of terminals
nonterms.asc	set of non-terminals
grammar.asc	production rules

- SLEN: An array containing entries for all the symbols in the grammar (terminals & non terminals). At the start it is initialized as:
  Non-terminals: set the value to infinity
  Terminals: set the value to 1 (will remain unchanged.)
  We start rewriting non-terminals using the production rule, which gives us the shortest length. At the end of the phase we get the shortest length of terminal string for each symbol.

  Length of string=No. of steps + No. of characters in the string
- RLEN: An array containing entries for each rule, it gives the length of the shortest terminal string which we get using that rule. Again, the length is the sum of the steps taken in the derivation and the number of terminals in the resulting string.
- SHORT: For each non-terminal we maintain an array containing the production number, which gives us the shortest terminal string.

We can check the grammar by the end of phase I. If any entry of SLEN is infinity or if SHORT contains –1, it is an indication that the grammar is ambiguous; there are some productions that are never used for deriving strings or some non-terminals have no rewritten rules (the grammar is incomplete). It is one of the unique methods to detect these errors in a context free grammar .

B. Phase II (Shortest Derivation Length)

Second phase uses the SLEN and RLEN computed in the previous phase and produces DLAN and PRE**,** to be used by the final phase.

Input: SLEN and RLEN

- DLEN: For each non-terminal, it gives the length of the shortest terminal string, used in its derivation.
- PREV: Contains the rule number use to introduce a non-terminal in the shortest terminal string derivation.

We calculate these two arrays for all non-terminals except the starting symbol. For the starting symbol the PREV should be –1, as it cannot be introduced by any rule and DLEN should be the same as SLEN.

At the end of this phase, DLEN should not be infinity (which in the programs is represented by the maximum possible integer value MAX_INIT) for any non-terminal and PREV should not be equal to –1 except for the Starting symbol. If this happens the grammar is erroneous.

C. Phase III (Generate Sentence)

In the third phase, the sentences for the given language are generated. First we push the start symbol on the stack and as long as the stack contains some elements, we keep on popping the top-most element and rewrite it using a production rule.

Input: SHORT and PREV

C1. Choose A Rule

The goal is to use each production rule at least once. In these implementations rules are selected on the basis of the values

in PREV and SHORT, whenever a rule with a low value of PREV or SHORT is found then we replace the existing one with it giving the minimum length sentences.

For a non-terminal A at the LHS, if a rule $\mathbf{A \rightarrow \alpha}$ exists, which has not yet used then we choose it. If more than one rule exists, then we choose the one with the lowest value of PREV and SHORT.

Else if a derivation $\mathbf{A ==> \alpha ==> \gamma_1 B \gamma_2}$ exists such that B is a non- terminal not on the stack and a rule $B \rightarrow \beta$ exists which has not been used, then use $A \rightarrow \alpha$ production which will then be rewritten using any of the α rules. [11]

For each non-terminal on the stack, we maintain the following arrays:

- ONST: Contains the occurrences of non-terminals on the stack .At the end it should be zero, their should not be any unused symbol on the stack.
- ONCE: For each non-terminal, an array is maintained such that it contains any one of the following values:
    1. READY: The production number previously in ONCE has been used and the next time this non-terminal will be rewritten using a different production.
    2. UNSURE: The value of ONCE calculated in the last loop is not sure.
    3. For some non-terminals it is the production number used to introduce that symbol in shortest string derivation and for some non-terminals this is not true.
    4. FINISHED: The non-terminal is rewritten using all possible productions and can't be used in any other way.
    5. INTEGER: Containing the production number used to rewrite the symbol in some useful derivation.

    In our implementation we have used an integer array for ONCE where the following values are represented as:

    Ready -1
    Unsure -2
    Finished -3
    From 0 to onwards are the rules numbers.
- MARK: For each rule it contains either true or false but at end of this phase all the entries should be true, which shows that each and every rule is used at least once, the main requirement of Purdom's algorithm.
- STACK: The stack should be empty at the end of this phase and all the symbols should be used in making sentences.

SLEN	**Non-terminals**	**Infinity**
	Terminals	**1**
RLEN	**Rule Number**	**Infinity**
SHORT	**Non-terminal**	**-1**

Table 5.1 Initialization of Phase-I

DLEN	**Infinity**
PREV	**-1**

Table 5.2 Initialization of Phase-II

ONST	**Zero**
ONCE	**READY**
MARK	**False**
STACK	**Push all the terminals and non terminals on to the stack**

Table 5.3 Initialization of Phase-II

Purdom's generated Sentences	MACS Test Cases	Syntactically Correct	Semantically Correct
VOID CLASSNAME_DOT ID_LP CLASSNAME ID COMMA CONST CLASSNAME ID RP SEMICOL	void A.a(A a1, const A b);	X	X
VOID CLASSNAME_DOT ID_LP BOOL ID RP SEMICOL	void A.a(bool b);	X	X
PUBLIC VOID CLASSNAME_DOT ID COMMA CLASSNAME_DOT ID SEMICOL	public void A.a,A.b;		

CLASS CLASSNAME SEMICOL	**class A;**	**X**	**X**
CLASS ID LB RB	**class a{}**	**X**	**X**
CLASS ID EXTENDS CLASSNAME SEMICOL	**class a extends A;**	**X**	**X**
CLASSNAME_DOT CLASSNAME_LP RP SEMICOL	**A.B( );**	**X**	**X**
MAIN_LP CONST STRING LS RS ID RP LB RB	**main(const string [] a){ }**	**X**	**X**
PUBLIC SHARED VOID CLASSNAME_DOT ID SEMICOL	**public shared void A.a;**	**X**	
PUBLIC CONST VOID CLASSNAME_DOT ID SEMICOL	**public const void A.a;**	**X**	
PRIVATE VOID CLASSNAME_DOT ID SEMICOL	**private void A.a;**	**X**	
PRIVATE SHARED VOID CLASSNAME_DOT ID SEMICOL	**private shared void A.a;**	**X**	
PRIVATE CONST VOID CLASSNAME_DOT ID SEMICOL	**private const void A.d;**	**X**	
SHARED VOID CLASSNAME_DOT ID SEMICOL	**shared void A.a;**	**X**	
SHARED PUBLIC VOID CLASSNAME_DOT ID SEMICOL	**shared public void A.a;**	**X**	
SHARED PRIVATE VOID CLASSNAME_DOT ID SEMICOL	**shared private void A.a;**	**X**	
SHARED CONST VOID CLASSNAME_DOT ID SEMICOL	**shared const void A.a;**	**X**	
CONST VOID CLASSNAME_DOT ID SEMICOL	**const void A.a;**	**X**	
CONST PUBLIC VOID CLASSNAME_DOT ID SEMICOL	**const public void A.a;**	**X**	
CONST PRIVATE VOID CLASSNAME_DOT ID SEMICOL	**const private void A.a;**	**X**	
CONST SHARED VOID CLASSNAME_DOT ID SEMICOL	**const shared void A.a;**	**X**	
MAIN_LP LB RB	**main({ })**	**X**	

Table 5.4 Sentences Generated By the Algorithm

VI. SUMMARY AND FUTURE RESEARCH

Although parser is one of the most important subset in compiler design, not much attention is given to parser design and especially to parser testing, there is a need to do more work in this area.

Purdom's algorithm is one of its kinds in generating test cases for parser; it is a complete method for testing small grammars. The test cases generated can be extended manually to incorporate some semantic aspects of the language for the complete validation of underlying compiler. However, more work is required to cover the language syntax and semantics in a systematic and formal way.

We implemented Purdom's algorithm using two different languages Java and C++ and have achieved the same results. The sentences generated are mostly syntactically and semantically correct with a few exceptions which are semantically incorrect. We have extended the test cases we get from the algorithm and combined the language semantics to validate both the static and dynamic aspects of the MACS grammar, providing the user with the guarantee that the MACS compiler is error free up to our best knowledge.

Our future research includes testing the most advanced features of MACS compilers to guarantee that it deals properly with more complex semantics features of the language such as data definitions parts of the program.

REFERENCES

[1] P. Purdom, "A Sentence Generator For Testing Parsers", BIT, vol 12:366-375, April 1972.

[2] A.S. Boujarwah and K. Saleh, "Compiler Test case generation methods: a survey and assessment", Information and software technology vol 39(9):617-625, May 1997.

[3] B. A. Malloy and J. F. Power , "Metric–Based Analysis of Context Free Grammars", Proceedings 8th International Workshop on Program Comprehension, IEEE Computer Society: Los Alamitos, CA, 171-178,2000.

[4] B. A. Malloy and J. T. Waldron, "Applying Software Engineering Techniques to Parser Design: The Development of a C# Parser", ACM International Conference Proceeding Series; Vol. 30:75-8,2002.

[5] B. A. Malloy and J. F. Power, "An interpretation of Purdom's algorithm for automatic generation of test cases", In 1st Annual International Conference on Computer and Information Science, Orlando, Florida, USA, October 3-5 2001.

[6] B. A. Malloy and J. F. Power, "A Top-down Presentation of Purdom's Sentence Generation Algorithm", National University of Ireland, Maynooth, 2005.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[8] J. Riehl, "Grammar Based Unit Testing for Parsers", Master's Thesis, University of Chicago, Dept. of Computer Science.

[9] S. Aamir, "Verification and Validation Aspects of Compiler Construction", Master's thesis, McMaster University, Dept. of Computing and Software, April 2007.

[10] A. Boujarwah and K. Saleh, "Compiler test suite: evaluation and use in an automated environment", Information and Software Technology vol 36 (10): 607-614, 1994.

[11] A. Celentano, S. Regizzi, P.D. Vigna, and C. Ghezzi, "Compiler testing using a sentence generator", Software- Practice and Experience, vol 10:897-913, June 1980.

[12] K.V. Hanford, "Automatic generation of test cases", IBM System Journal. 242-258,1970.

[13] C.J. Burgess, and M. Saidi, "The Automatic Generation of Test Cases for Optimizing Fortran Compilers", Information Software Technology, vol 38:111-119., 1996

[14] J. B. Goodenough, "The Ada Compiler Validation Capacity", 1981.

[15] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing", IEEE Transactions on Software Engineering, SE-8(4):343-353, July 1982

APPENDIX - MACS LL(1) GRAMMAR

The usual yacc or bison notation is used, the tokens are the names completely in upper-case letters (e.g. MAIN_LP), the non-terminals are the names with first latter capitalized, but otherwise in lower-case letters (e.g. Program). Empty rules (as Program: ;) represent the so-called epsilon rules (i.e. Program).

Program: MAIN_LP MainSection Program1;
Program: CLASSNAME_DOT ConstrDecl Program1;
Program: CLASS ClassDeclDef Program1;
Program: PUBLIC Prefix10 AttrMethodDecl Program1;
Program: PRIVATE Prefix20 AttrMethodDecl Program1;
Program: SHARED Prefix30 AttrMethodDecl Program1;
Program: CONST Prefix40 AttrMethodDecl Program1;
Program: BOOL ArrayDim AttrMethodDecl1 Program1;
Program: CHAR ArrayDim AttrMethodDecl1 Program1;
Program: FLOAT ArrayDim AttrMethodDecl1 Program1;
Program: INT ArrayDim AttrMethodDecl1 Program1;
Program: STRING ArrayDim AttrMethodDecl1 Program1;
Program: CLASSNAME ArrayDim AttrMethodDecl1 Program1;
Program: VOID CLASSNAME_DOT ID_LP Args SEMICOL;
Program: ;
Program1: MAIN_LP MainSection Program1;
Program1: CLASSNAME_DOT ConstrDecl Program1;
Program1: CLASS ClassDeclDef Program1;
Program1: PUBLIC Prefix10 AttrMethodDecl Program1;
Program1: PRIVATE Prefix20 AttrMethodDecl Program1;
Program1: SHARED Prefix30 AttrMethodDecl Program1;
Program1: CONST Prefix40 AttrMethodDecl Program1;
Program1: BOOL ArrayDim AttrMethodDecl1 Program1;
Program1: CHAR ArrayDim AttrMethodDecl1 Program1;
Program1: FLOAT ArrayDim AttrMethodDecl1 Program1;
Program1: INT ArrayDim AttrMethodDecl1 Program1;
Program1: STRING ArrayDim AttrMethodDecl1 Program1;
Program1: CLASSNAME ArrayDim AttrMethodDecl1 Program1;
Program1: VOID CLASSNAME_DOT ID_LP Args SEMICOL;
Program1: ;
MainSection: LB MethodBody;
MainSection: CONST STRING LS RS ID RP LB MethodBody;
AttrMethodDecl: DataType AttrMethodDecl1;
AttrMethodDecl1: CLASSNAME_DOT AttrMethodDecl2;
AttrMethodDecl2: ID_LP Args SEMICOL;
AttrMethodDecl2: ID AttrDecl;
AttrDecl: SEMICOL;
AttrDecl: COMMA CLASSNAME_DOT ID AttrDecl;
ConstrDecl: CLASSNAME_LP Args SEMICOL;
ClassDeclDef: ID ClassDeclDef1;
ClassDeclDef: CLASSNAME ClassDeclDef1;
ClassDeclDef1: EXTENDS CLASSNAME ClassDeclDef2;
ClassDeclDef1: LB ClassBody;
ClassDeclDef1: SEMICOL;
ClassDeclDef2: LB ClassBody;
ClassDeclDef2: SEMICOL;
ClassBody: ClassMember ClassBody;
ClassBody: RB;
ClassMember: CLASSNAME_LP ConstrDef;
ClassMember: Prefix DataType AttrMethodDef;
DataType: BOOL ArrayDim;
DataType: CHAR ArrayDim;
DataType: FLOAT ArrayDim;
DataType: INT ArrayDim;
DataType: STRING ArrayDim;
DataType: CLASSNAME ArrayDim;
DataType: VOID;
ArrayDim: LS RS ArrayDim;
ArrayDim: ;
Prefix: PUBLIC Prefix10;
Prefix: PRIVATE Prefix20;
Prefix: SHARED Prefix30;
Prefix: CONST Prefix40;
Prefix: ;
Prefix10: SHARED Prefix13;
Prefix10: CONST Prefix14;
Prefix10: ;
Prefix13: CONST;
Prefix13: ;
Prefix14: SHARED;
Prefix14: ;
Prefix20: SHARED Prefix23;
Prefix20: CONST Prefix24;
Prefix20: ;
Prefix23: CONST;
Prefix23: ;
Prefix24: SHARED;
Prefix24: ;
Prefix30: PUBLIC Prefix31;
Prefix30: PRIVATE Prefix32;
Prefix30: CONST Prefix34;
Prefix30: ;
Prefix31: CONST;
Prefix31: ;
Prefix32: CONST;
Prefix32: ;
Prefix34: PUBLIC;
Prefix34: PRIVATE;
Prefix34: ;
Prefix40: PUBLIC Prefix41;
Prefix40: PRIVATE Prefix42;
Prefix40: SHARED Prefix43;
Prefix40: ;
Prefix41: SHARED;
Prefix41: ;
Prefix42: SHARED;
Prefix42: ;
Prefix43: PUBLIC;
Prefix43: PRIVATE;
Prefix43: ;
AttrMethodDef: ID Init AttrDef;
AttrMethodDef: ID_LP Args LB MethodBody;
AttrDef: COMMA ID Init AttrDef;
AttrDef: SEMICOL;
ConstrDef: Args LB ParentConstr MethodBody;
ParentConstr: PARENT_LP Params SEMICOL;
ParentConstr: ;
PassingSpec: AND;
PassingSpec: ;

ArgType: CONST ArgType1 PassingSpec;
ArgType: BOOL ArrayDim PassingSpec;
ArgType: CHAR ArrayDim PassingSpec;
ArgType: FLOAT ArrayDim PassingSpec;
ArgType: INT ArrayDim PassingSpec;
ArgType: STRING ArrayDim PassingSpec;
ArgType: CLASSNAME ArrayDim PassingSpec;
ArgType1: BOOL ArrayDim;
ArgType1: CHAR ArrayDim;
ArgType1: FLOAT ArrayDim;
ArgType1: INT ArrayDim;
ArgType1: STRING ArrayDim;
ArgType1: CLASSNAME ArrayDim;
Args: RP;
Args: ArgType ID Args1;
Args1: RP;
Args1: COMMA ArgType ID Args1;
VarDef1: CONST VarDef11;
VarDef1: BOOL ArrayDim VarDef12;
VarDef1: CHAR ArrayDim VarDef12;
VarDef1: FLOAT ArrayDim VarDef12;
VarDef1: INT ArrayDim VarDef12;
VarDef1: STRING ArrayDim VarDef12;
VarDef1: CLASSNAME ArrayDim VarDef12;
VarDef2: PERM VarDef21;
VarDef2: BOOL ArrayDim VarDef22;
VarDef2: CHAR ArrayDim VarDef22;
VarDef2: FLOAT ArrayDim VarDef22;
VarDef2: INT ArrayDim VarDef22;
VarDef2: STRING ArrayDim VarDef22;
VarDef2: CLASSNAME ArrayDim VarDef22;
VarDef3: ID Init VarDef4;
VarDef4: COMMA ID Init VarDef4;
VarDef4: ;
VarDef11: BOOL ArrayDim VarDef111;
VarDef11: CHAR ArrayDim VarDef111;
VarDef11: FLOAT ArrayDim VarDef111;
VarDef11: INT ArrayDim VarDef111;
VarDef11: STRING ArrayDim VarDef111;
VarDef11: CLASSNAME ArrayDim VarDef111;
VarDef21: BOOL ArrayDim VarDef211;
VarDef21: CHAR ArrayDim VarDef211;
VarDef21: FLOAT ArrayDim VarDef211;
VarDef21: INT ArrayDim VarDef211;
VarDef21: STRING ArrayDim VarDef211;
VarDef21: CLASSNAME ArrayDim VarDef211;
VarDef12: ID Init VarDef4;
VarDef22: ID Init VarDef4;
VarDef111: ID Init VarDef4;
VarDef211: ID Init VarDef4;
Init: ASSIG Expr;
Init: PASSIG Expr;
Init: ;
MethodBody: ID_COLON UStm MethodBody;
MethodBody: SEMICOL MethodBody;
MethodBody: IF IfStm MethodBody;
MethodBody: FOR ForStm MethodBody;
MethodBody: WHILE WhileStm MethodBody;
MethodBody: GOTO ID SEMICOL MethodBody;
MethodBody: CONTINUE SEMICOL MethodBody;
MethodBody: BREAK SEMICOL MethodBody;
MethodBody: TERMINATE SEMICOL MethodBody;
MethodBody: RETURN ReturnStm MethodBody;
MethodBody: PRETURN Expr SEMICOL MethodBody;
MethodBody: THROW Expr SEMICOL MethodBody;
MethodBody: TRY LB MethodBody CATCH LP Catch MethodBody;
MethodBody: PERM VarDef1 MethodBody;
MethodBody: CONST VarDef2 MethodBody;
MethodBody: BOOL ArrayDim VarDef3 MethodBody;
MethodBody: CHAR ArrayDim VarDef3 MethodBody;
MethodBody: FLOAT ArrayDim VarDef3 MethodBody;
MethodBody: INT ArrayDim VarDef3 MethodBody;
MethodBody: STRING ArrayDim VarDef3 MethodBody;
MethodBody: CLASSNAME ArrayDim VarDef3 MethodBody;
MethodBody: RB;
Catch: ID RP Catch1;
Catch1: LB MethodBody;
Stm: ID_COLON UStm;
Stm: UStm;
UStm: SEMICOL;
UStm: IF IfStm;
UStm: FOR ForStm;
UStm: WHILE WhileStm;
UStm: GOTO ID SEMICOL;
UStm: CONTINUE SEMICOL;
UStm: BREAK SEMICOL;
UStm: TERMINATE SEMICOL;
UStm: RETURN ReturnStm;
UStm: PRETURN Expr SEMICOL;
UStm: THROW Expr SEMICOL;
UStm: TRY LB MethodBody CATCH LP Catch;
UStm: Expr SEMICOL;
IfStm: LP Cond ThenBlock IfStm1;
IfStm1: ELSE Block;
IfStm1: PEEKNOTELSE;
ThenBlock: LB ThenBlock1;
ThenBlock: UStm;
ThenBlock1: RB;
ThenBlock1: Stm ThenBlock1;
Block: LB Block1;
Block: UStm;
Block1: RB;
Block1: Stm Block1;
ForStm: LP Prolog Cond1 Epilog Block;
Cond1: SEMICOL;
Cond1: Expr SEMICOL;
Prolog: SEMICOL;
Prolog: Expr Prolog1;
Prolog1: SEMICOL;
Prolog1: COMMA Expr Prolog1;
Epilog: RP;
Epilog: Expr Epilog1;
Epilog1: RP;
Epilog1: COMMA Expr Epilog1;
Cond: RP;
Cond: Expr RP;
WhileStm: LP Cond Block;
ReturnStm: SEMICOL;

ReturnStm: Expr SEMICOL;
Expr: Factor;
Factor: Term Factor1;
Factor1: PLUS SimpleExpr Factor1;
Factor1: MINUS SimpleExpr Factor1;
Factor1: LT SimpleExpr Factor1;
Factor1: LE SimpleExpr Factor1;
Factor1: LTLT SimpleExpr Factor1;
Factor1: GT SimpleExpr Factor1;
Factor1: GE SimpleExpr Factor1;
Factor1: GTGT SimpleExpr Factor1;
Factor1: AND SimpleExpr Factor1;
Factor1: OR SimpleExpr Factor1;
Factor1: EQ SimpleExpr Factor1;
Factor1: NEQ SimpleExpr Factor1;
Factor1: MOD SimpleExpr Factor1;
Factor1: ASSIG Expr;
Factor1: PASSIG Expr;
Factor1: ;
Term: SimpleExpr Term1;
Term1: STAR SimpleExpr Term1;
Term1: SLASH SimpleExpr Term1;
Term1: ;
SimpleExpr: LP CastOrPexp;
SimpleExpr: PLUS SimpleExpr1;
SimpleExpr: MINUS SimpleExpr1;
SimpleExpr: NOT SimpleExpr1;
SimpleExpr: SIZEOF SimpleExpr1;
SimpleExpr: TYPEOF SimpleExpr1;
SimpleExpr: IDOF SimpleExpr1;
SimpleExpr: PLUSPLUS SimpleExpr1;
SimpleExpr: MINUSMINUS SimpleExpr1;
SimpleExpr: Ref;
SimpleExpr1: FALSE;
SimpleExpr1: TRUE;
SimpleExpr1: NOREF;
SimpleExpr1: CHAR_LIT;
SimpleExpr1: FLOAT_LIT;
SimpleExpr1: INT_LIT;
SimpleExpr1: STRING_LIT StringLit;
SimpleExpr1: Ref;
StringLit: LS Expr StringLit1;
StringLit: ;
StringLit1: RS;
StringLit1: COLON Expr RS StringLit2;
StringLit2: LS Expr RS;
StringLit2: ;
CastOrPexp: BOOL RP SimpleExpr1;
CastOrPexp: CHAR RP SimpleExpr1;
CastOrPexp: FLOAT RP SimpleExpr1;
CastOrPexp: INT RP SimpleExpr1;
CastOrPexp: STRING RP SimpleExpr1;
CastOrPexp: CLASSNAME_RP SimpleExpr1;
CastOrPexp: Expr RP Ref3;
Ref: CLASSNAME_DOT Ref1;
Ref: PARENT_DOT Ref1;
Ref: Ref2;
Ref1: PARENT_DOT Ref1;
Ref1: Ref2;
Ref2: ID Ref3;
Ref2: ID_LP Params Ref3;
Ref3: DOT Ref2;
Ref3: LS Ref4;
Ref3: PLUSPLUS;
Ref3: MINUSMINUS;
Ref3: ;
Ref4: RS Ref3;
Ref4: Expr Ref5;
Ref5: RS Ref3;
Ref5: COLON Expr RS Ref6;
Ref6: LS Expr RS;
Ref6: ;
Params: RP;
Params: Expr Params1;
Params1: RP;
Params1: COMMA Expr Params1;