Software Design Document

Lina Vo
CSC 450
April 25, 2016

Table of Contents

1 Introduction

The purpose of this project is to create a compiler that takes in a mini Pascal language as the source language and translates it into MIPS Assembly as the target language. The compiler is programmed in Java *(see figure 1.0)*. This project is the senior keystone at Augsburg College that spanned over the course of an entire school year.

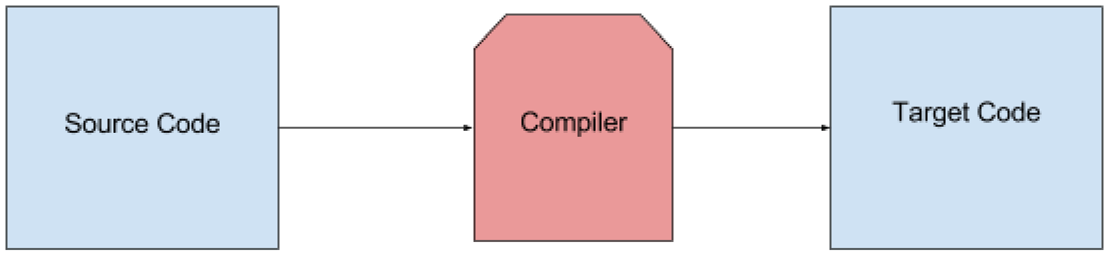


Figure 1.0

The source code in this case is the mini Pascal aforementioned, the compiler that is written in Java then takes in the text file, parses it, builds the corresponding tree, then with the proper tree, the compiler will generate MIPS assembly code.

2 Overviews

2.1 Scanner

The lexical scanner is responsible for collecting lexemes from a text file and making them available one at at time. By having the lexemes available individually, they may be passed to the parser component of the compiler. Lexemes are meaningful syntactic units. The scanner is heavily influenced by the deterministic finite automata (*page 9*) and in turn the deterministic finite automata has been influenced by the grammar as well as the lexemes list (*page 10*).

2.2 Recognizer

The purpose of the recognizer is to take the tokens given from the scanner and interprets them. It then interprets whether the inputted mini pascal language is valid or not. If it is valid then it will compile if not then it will throw an error. The recognizer was implemented based on the grammar (*starting on page 12)* given in class.

2.3 Parser

The purpose of the parser is to parse through the pascal code and create a syntax tree by building out various nodes. The parser is to follow the grammar handed out in class and listed starting on *page 12*.

2.4 Syntax Tree

The syntax tree is built out from the parser and is comprised of various nodes that represent segments of the grammar (*see page 12)*. The UML diagram for the syntax tree can be seen on *page 7*. The syntax tree is built with recursive descent and is LL1.

2.2 Code generation

The code generation takes the syntax tree produced from the parser and then produces the corresponding MIPS Assembly code by taking individual nodes and then building out the correct MIPS Assembly that is correlated to that specific node in the tree.

3 Scanner

The scanner works by taking in an input which is handled with a PushBackReader. The Pushback reader allows for any unnecessary characters to be pushed back into the input stream. The Scanner includes various components. The components include tokens that are enums, the PushBackReader, and a TransitionTable that is a hashmap that has keywords and symbols based on the lexemes list (*page 6*). The lexeme is built up by having characters appended one by one until the token is completed and checked against the TransitionTable. As the lexeme is being appended character by character the scanner will move to the proper corresponding states based on the NextToken(). For example, the following line of code would be processed like so.

```
integer myAccount = 5;
```

The scanner would first recognize the `i` in integer as a letter and move to the IN_ID_OR_KEYWORD state and continue to repeat this process until it completes appending `r` in integer. When it sees the white space the state is changed to ID_OR_KEYWORD_COMPLETE state, integer is then checked against the keyword list in the TransitionTable to see if it is recognized as a keyword. Since it is in the list it is recognized as a keyword. The scanner proceeds to move on to the `m` in `myAccount`. It sees a letter so it moves to the IN_ID_OR_KEYWORD state and continues this process until it completes appending all the letters to the currentLexeme. Once again, when the scanner sees white space, the state is changed to ID_OR_KEYWORD_COMPLETE and then checked against the list of keywords in the TransitionTable. Since `myAccount` was not on the list, it is recognized as an ID. The scanner proceeds to see the `=` and goes to the SHORT_SYMBOL_COMPLETE state and appends it to currentLexeme. Once it sees white space again, the scanner checks to see if the `=` is on the symbols list in the TransitionTable. Since it is, the scanner recognizes it as a legal symbol in the mini Pascal language and continues. The scanner then recognizes a digit `5` and proceeds to the DIGIT state. Since the scanner recognizes something that is not an `E` or a `.` it continues to the NUMBER_COMPLETE state and is recognized as an INTEGER since it did not have `E` or `.` next. Lastly, the scanner sees the `;` and goes to the SHORT_SYMBOL_COMPLETE state and appends it to currentLexeme. Once it sees white space again, the scanner checks to see if the `=` is on the symbols list in the TransitionTable. Since it is, the scanner recognizes it as a legal symbol in the mini Pascal language and continues. The scanner then recognizes that it is the end of the file and stops gracefully.

4 Recognizer

The recognizer will eventually be a completed parser. It works by taking in a token at a time from the scanner and seeing if it matches the grammar. The recognizer was implemented based on the grammar given in class. If the inputted mini pascal program is valid then no error will be thrown. If it does not match up with the grammar then an error is thrown. For example, see below…

```
compound_statement ->      begin optional_statements end
```

The recognizer would make sure that “begin” was present before it calls the optional_statments() function. If “begin” was not present then an error would be thrown. After the optional_statements have been called and assuming that you didn’t run into any syntax errors while parsing the optional statements in the file the recognizer would check to make sure that the lexeme “end” was present. If it is not present an error would be thrown.

5 Parser

The parser is the joining point of the recognizer and building a syntax tree. Once the recognizer is built out and the syntax tree has been implemented. The parser iterates through the text file and then builds the corresponding nodes based on grammar. The parser is a very important part of the compiler as the syntax tree produced by it will be used later to create the proper MIPS Assembly code.

6 Syntax Tree

The syntax tree is built from the parser. The syntax tree in this specific compiler was based on the UML diagram in *figure 6.0*.

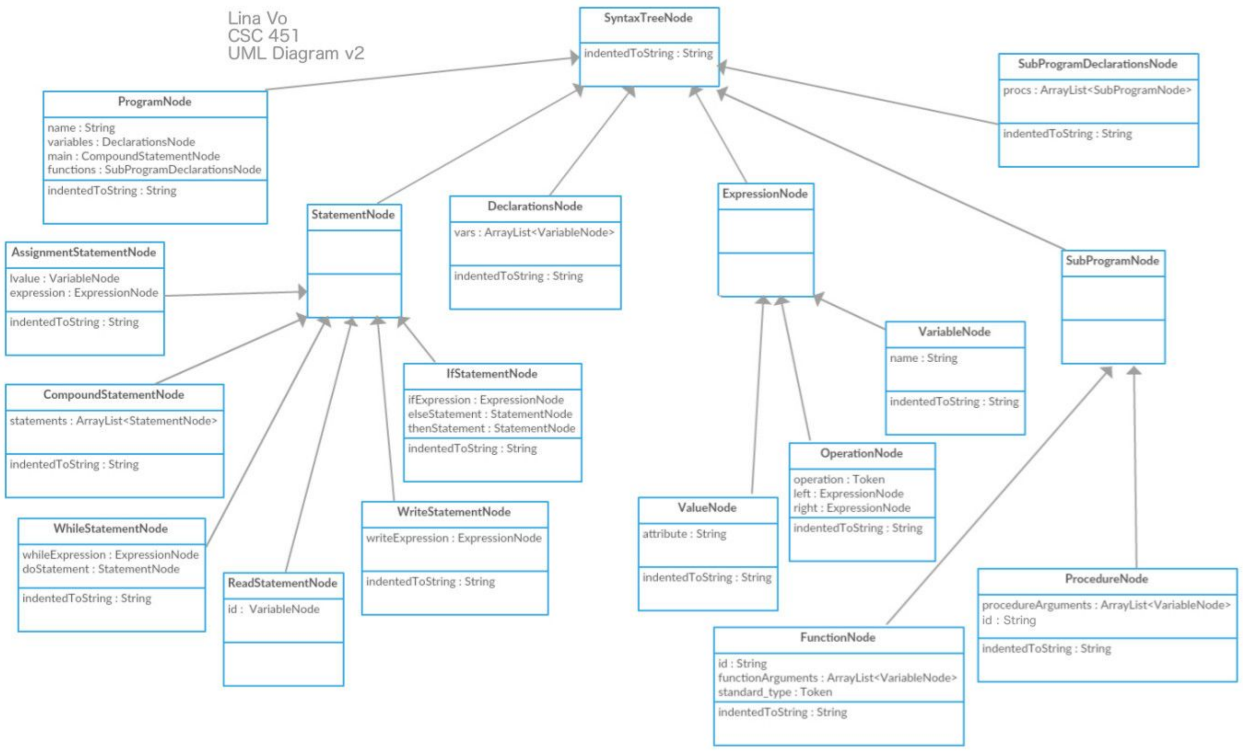

Lina Vo
CSC 451
UML Diagram v2
SyntaxTreeNode
indentedToString : String
ProgramNode
name : String
variables : DeclarationsNode
main : CompoundStatementNode
functions : SubProgramDeclarationsNode
indentedToString : String
SubProgramDeclarationsNode
procs : ArrayList<SubProgramNode>
indentedToString : String
StatementNode
DeclarationsNode
vars : ArrayList<VariableNode>
indentedToString : String
ExpressionNode
SubProgramNode
AssignmentStatementNode
lvalue : VariableNode
expression : ExpressionNode
indentedToString : String
VariableNode
name : String
indentedToString : String
CompoundStatementNode
statements : ArrayList<StatementNode>
indentedToString : String
IfStatementNode
ifExpression : ExpressionNode
elseStatement : StatementNode
thenStatement : StatementNode
indentedToString : String
OperationNode
operation : Token
left : ExpressionNode
right : ExpressionNode
indentedToString : String
WhileStatementNode
whileExpression : ExpressionNode
doStatement : StatementNode
indentedToString : String
WriteStatementNode
writeExpression : ExpressionNode
indentedToString : String
ValueNode
attribute : String
indentedToString : String
ReadStatementNode
id : VariableNode
ProcedureNode
procedureArguments : ArrayList<VariableNode>
id : String
indentedToString : String
FunctionNode
id : String
functionArguments : ArrayList<VariableNode>
standard_type : Token
indentedToString : String

figure 6.0

For example, see demonstration below…

```
var foo : integer;
```

If the parser were to look at this specific line of a pascal code from a text file, the parser would recognize that this was a `DeclartionsNode`. `DeclarationsNode` is made up of an ArrayList of `VariableNodes`. So the parser would create a `VariableNode` that extends Expression node that has the information "foo" as a string stored and the value "integer" stored.

7 Code Generation

Code generation works by taking the completed syntax tree, taking the corresponding nodes created, and then translating those nodes into the proper MIPS assembly. See the example below.

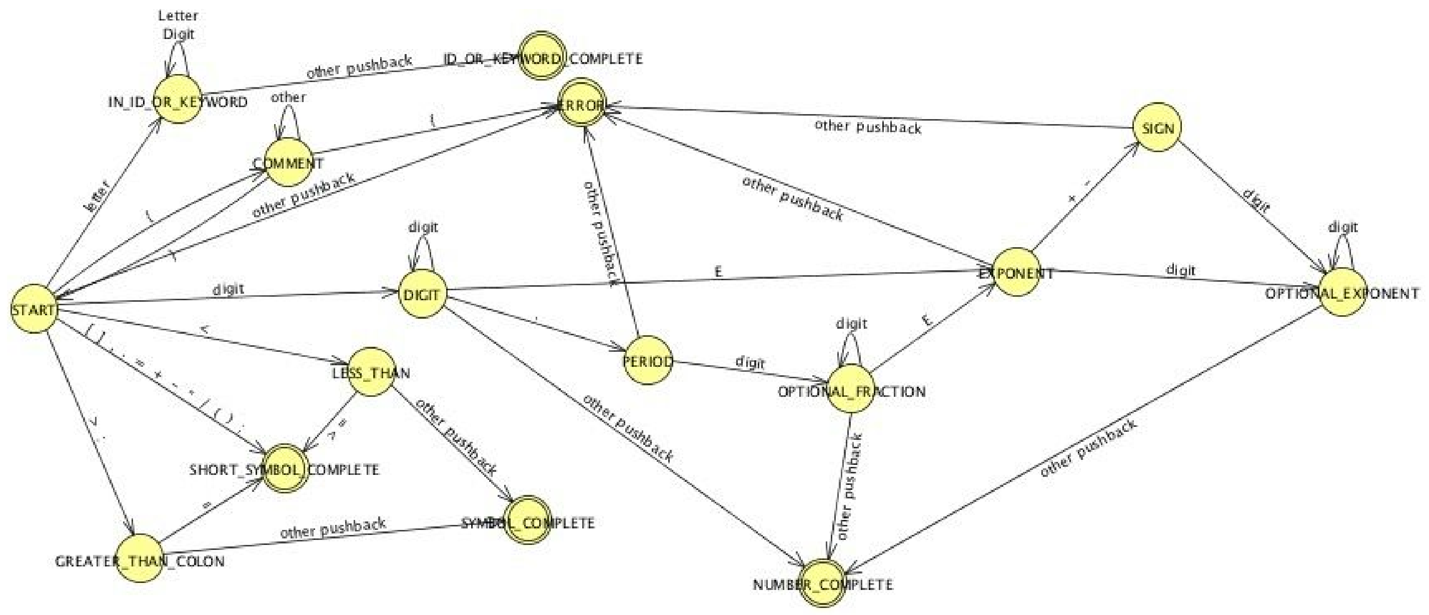
```
foo := 5 + 2;
```

There is a method in the code generation class there is a function called `minus(int rs, int rt, int store)` that gets called when an `OperationNode` is seen. In specific example, store would be the register at which `foo`'s value is stored which is `currentRegsiter - 1`. The other integers are stored in `currentRegister` and `currentRegister - 1`. Suppose `currentRegister` is 1. The corresponding subtraction instruction in assembly is `sub $d, $rt, $rs` The function would append various components to print out the following assembly code.

```
sub $t0, $t0, $t1
```

8 Deterministic Finite Automata

This image is a product of the grammar given for the mini Pascal language as well as the lexemes list seen on *page 6*.

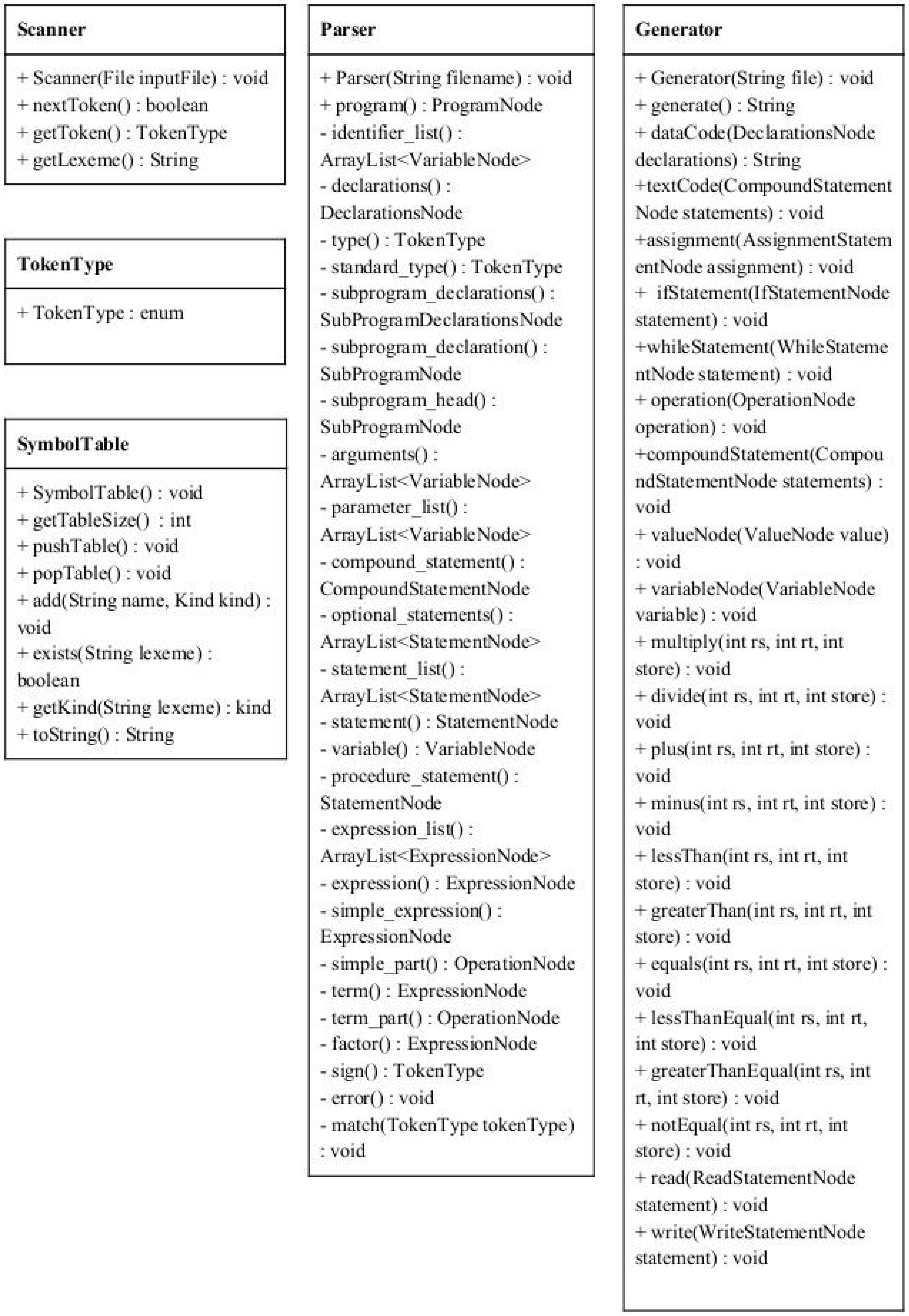


9 Lexemes List

Keywords	Symbols
1. or	1. =
2. div	2. >
3. mod	3. <
4. and	4. < >
5. var	5. <=
6. program	6. >=
7. integer	7. +
8. array	8. -
9. of	9. *
10. real	10. /
11. function	11. :=
12. procedure	12. {
13. begin	13. }
14. end	14. (
15. if	15.)
16. then	16. [
17. else	17.]
18. do	18. ;
19. while	19. :
20. not	20. .
	21. E
	22. ,

10 UML Diagram for Compiler

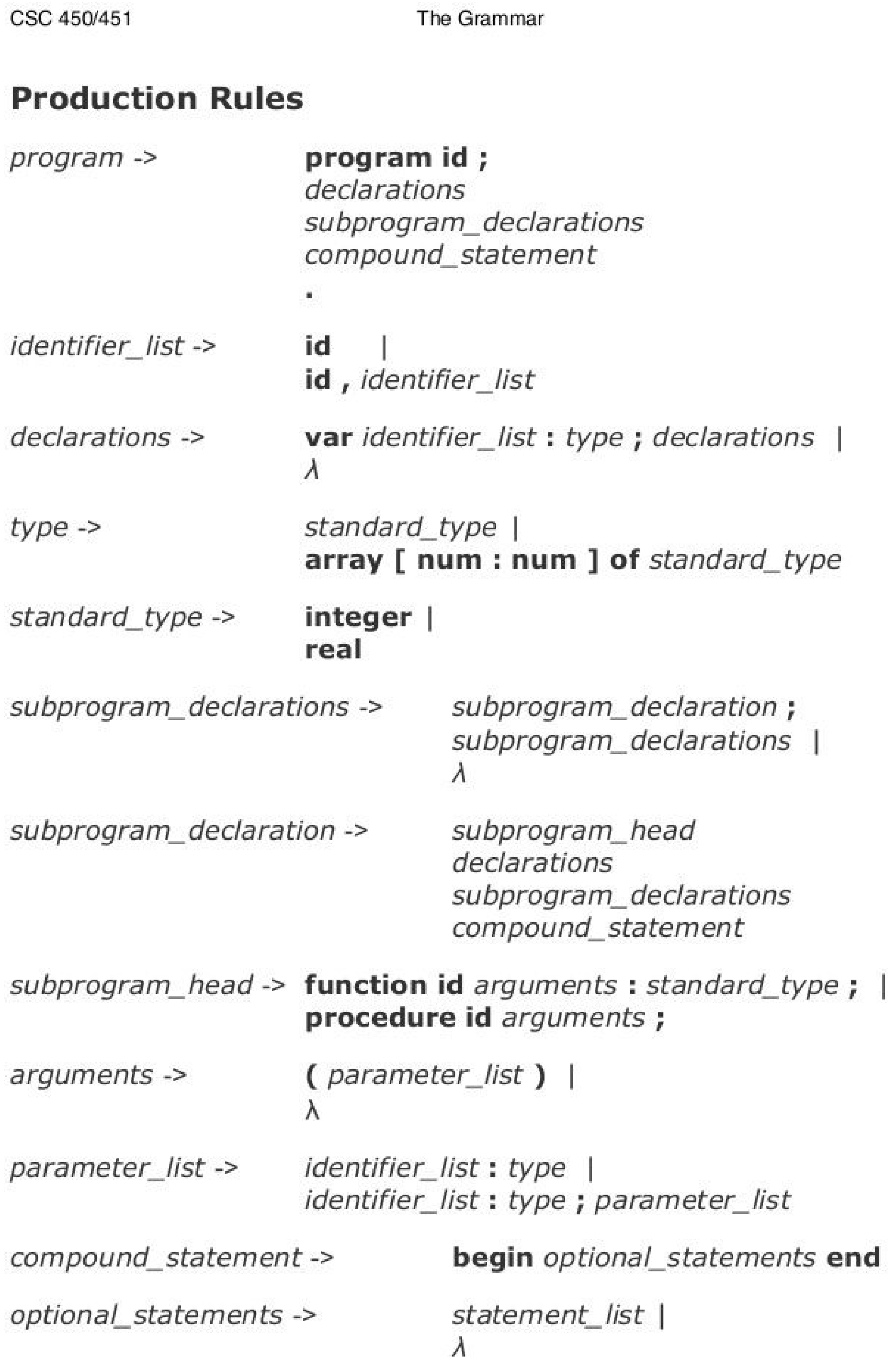
Note that the following diagram is not including the UML diagram listed for the syntax tree (*see page 7, figure 6.0*) and is for the remainder of the compiler.



Scanner
+ Scanner(File inputFile) : void
+ nextToken() : boolean
+ getToken() : TokenType
+ getLexeme() : String
TokenType
+ TokenType : enum
SymbolTable
+ SymbolTable() : void
+ getTableSize() : int
+ pushTable() : void
+ popTable() : void
+ add(String name, Kind kind) : void
+ exists(String lexeme) : boolean
+ getKind(String lexeme) : kind
+ toString() : String
Parser
+ Parser(String filename) : void
+ program() : ProgramNode
- identifier_list() : ArrayList<VariableNode>
- declarations() : DeclarationsNode
- type() : TokenType
- standard_type() : TokenType
- subprogram_declarations() : SubProgramDeclarationsNode
- subprogram_declaration() : SubProgramNode
- subprogram_head() : SubProgramNode
- arguments() : ArrayList<VariableNode>
- parameter_list() : ArrayList<VariableNode>
- compound_statement() : CompoundStatementNode
- optional_statements() : ArrayList<StatementNode>
- statement_list() : ArrayList<StatementNode>
- statement() : StatementNode
- variable() : VariableNode
- procedure_statement() : StatementNode
- expression_list() : ArrayList<ExpressionNode>
- expression() : ExpressionNode
- simple_expression() : ExpressionNode
- simple_part() : OperationNode
- term() : ExpressionNode
- term_part() : OperationNode
- factor() : ExpressionNode
- sign() : TokenType
- error() : void
- match(TokenType tokenType) : void
Generator
+ Generator(String file) : void
+ generate() : String
+ dataCode(DeclarationsNode declarations) : String
+textCode(CompoundStatementNode statements) : void
+assignment(AssignmentStatementNode assignment) : void
+ ifStatement(IfStatementNode statement) : void
+whileStatement(WhileStatementNode statement) : void
+ operation(OperationNode operation) : void
+compoundStatement(CompoundStatementNode statements) : void
+ valueNode(ValueNode value) : void
+ variableNode(VariableNode variable) : void
+ multiply(int rs, int rt, int store) : void
+ divide(int rs, int rt, int store) : void
+ plus(int rs, int rt, int store) : void
+ minus(int rs, int rt, int store) : void
+ lessThan(int rs, int rt, int store) : void
+ greaterThan(int rs, int rt, int store) : void
+ equals(int rs, int rt, int store) : void
+ lessThanEqual(int rs, int rt, int store) : void
+ greaterThanEqual(int rs, int rt, int store) : void
+ notEqual(int rs, int rt, int store) : void
+ read(ReadStatementNode statement) : void
+ write(WriteStatementNode statement) : void

11 Grammar

Attached is the grammar given in class.



CSC 450/451 The Grammar

Production Rules

program ->	**program id ;** *declarations* *subprogram_declarations* *compound_statement* **.**
identifier_list ->	**id** | **id ,** *identifier_list*
declarations ->	**var** *identifier_list* **:** *type* **;** *declarations* | *λ*
type ->	*standard_type* | **array [num : num] of** *standard_type*
standard_type ->	**integer** | **real**
subprogram_declarations ->	*subprogram_declaration* **;** *subprogram_declarations* | *λ*
subprogram_declaration ->	*subprogram_head* *declarations* *subprogram_declarations* *compound_statement*
subprogram_head ->	**function id** *arguments* **:** *standard_type* **;** | **procedure id** *arguments* **;**
arguments ->	**(** *parameter_list* **)** | *λ*
parameter_list ->	*identifier_list* **:** *type* | *identifier_list* **:** *type* **;** *parameter_list*
compound_statement ->	**begin** *optional_statements* **end**
optional_statements ->	*statement_list* | *λ*

statement_list ->	*statement* | *statement* **;** *statement_list*
statement ->	*variable* **assignop** *expression* | *procedure_statement* | *compound_statement* | **if** *expression* **then** *statement* **else** *statement* | **while** *expression* **do** *statement* | *read (id)* | *write (expression)*
variable ->	**id** | **id [** *expression* **]**
procedure_statement ->	**id** | **id (** *expression_list* **)**
expression_list ->	*expression* | *expression* **,** *expression_list*
expression ->	*simple_expression* | *simple_expression* **relop** *simple_expression*
simple_expression ->	*term simple_part* | *sign term simple_part*
simple_part ->	**addop** *term simple_part* | *λ*
term ->	*factor term_part*
term_part ->	**mulop** *factor term_part* | *λ*
factor ->	**id** | **id [** *expression* **]** | **id (** *expression_list* **)** | **num** | **(** *expression* **)** | **not** *factor*
sign ->	**+** | **-**

Lexical Conventions

1. Comments are surrounded by **{** and **}**. They may not contain a {. Comments may appear after any token.

2. Blanks between tokens are optional.

3. Token **id** for identifiers matches a letter followed by letter or digits:
   letter -> [a-zA-Z]
   digit -> [0-9]
   id -> letter (letter | digit)*

The ***** indicates that the choice in the parentheses may be made as many times as you wish.

1. Token **num** matches numbers as follows:
   digits -> digit digit*
   optional_fraction -> . digits | λ
   optional_exponent -> (E (+ | - | λ) digits) | λ
   num -> digits optional_fraction optional_exponent

2. Keywords are reserved.

3. The relational operators (**relop**'s) are:
   =, **<>**, **<**, **<=**, **>=**, and **>**.

4. The **addop**'s are **+**, **-**, and **or**.

5. The **mulop**'s are *****, **/**, **div**, **mod**, and **and**.

6. The lexeme for token **assignop** is **:=**.