BARZ Compiler: An Overview of the Syntactic Analyzer Implementation

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Syntax analysis is a crucial step for completing the front-end phase of conventional compilers. It is the process of transforming a token sequence into a structured representation of the program. This process utilizes a context-free grammar (CFG) to parse the input and generate an abstract syntax tree (AST). The AST serves as an intermediate representation that captures the syntactic structure of the code. This tree-based structure replaces the linear token sequence and provides a representation for subsequent compiler phases to perform further analysis, augmentation, and transformation. In this report, I present the design and implementation of the syntax analyzer for the BARZ compiler. Firstly, a context-free grammar written with Extended Backus–Naur Form (EBNF) notations is converted to a right-recursive list-generating productions. Then using the UCalgary tool, ambiguities are identified and these ambiguities are removed using factorization and alterations to the original grammar. The transformed grammar is a CFG LL(1) grammar that is used to generate FIRST and FOLLOW sets. These sets are then used to generate a parsing table which is used to implement a table-driven predictive parser. This report discusses the tools, libraries, and techniques used in the analysis and implementation, justifying the choices made.**

***Index Terms*—Compiler Design, Syntax Analyzer, LL(1) Grammar**

I. INTRODUCTION

A linear sequence representation of tokens that are generated by the lexical analyzer generally does not possess a syntactically meaningful structure. The syntax analyzer, usually denoted by `Parser`, is the program that takes in a token stream from the lexical analyzer and parses it according to a given grammar **G**. Grammar **G** is usually defined as `G = (N, T, S, R)`, where `N` is Nonterminal Symbols, `T` is Terminal Symbols, `S` is Starting Symbol, and `R` is Grammar Rules. Using this grammar definition, a predictive parser can be implemented with proper error recovery methods.

In this work, a grammar is given for an arbitrary programming language that is in Extended Backus–Naur Form (EBNF); the grammar is listed in the Syntactical Specifications. This grammar contains EBNF-style repetition and optionality notations and there are left-recursive and non-recursive ambiguities. These EBNF-style notations and left-recursive ambiguities are removed using the `grammartool.jar` provided by Dr. Joey Paquet. Then, the tool is used to transform the grammar into a form that can be recognized by the UCalgary tool [https://smlweb.cpsc.ucalgary.ca/start.html]. The UCalgary tool is used to detect ambiguities and these ambiguities are then further removed by applying factorizations and transformations to the grammar. The resulting grammar is a CFG LL(1) non-ambiguous grammar that can be used to implement a predictive parser.

Once the grammar rules/productions are finalized, the FIRST and FOLLOW sets are generated using the UCalgary tool as well as a LL(1) parsing table. This table is then used to implement a table-driven predictive parser. For the portion of this work, the predictive parser records derivations of the program from the starting symbol and outputs it to a derivation file. It also implements an error-recovery strategy and documents the errors encountered.

This is submitted as part of COMP-6421 Assignment #2.

II. SYNTACTICAL SPECIFICATIONS

The syntax of the grammar is defined as a 4-tuple $G = (N, T, S, R)$ where:

A. *N - Nonterminal Symbols*

`START`, `aParams`, `aParamsTail`, `addOp`, `arithExpr`, `arraySize`, `assignOp`, `assignStat`, `attributeDecl`, `classDecl`, `classOrImplOrFunc`, `expr`, `fParams`, `fParamsTail`, `factor`, `funcBody`, `funcDecl`, `funcDef`, `funcHead`, `functionCall`, `idnest`, `idOrSelf`, `implDef`, `indice`, `localVarDecl`, `localVarDeclOrStat`, `memberDecl`, `multOp`, `prog`, `relExpr`, `relOp`, `returnType`, `sign`, `statBlock`, `statement`, `term`, `type`, `varDecl`, `variable`, `visibility`

B. *T - Terminal Symbols*

`,`, `+`, `-`, `or`, `[`, `intLit`, `]`, `:=`, `class`, `id`, `{`, `}`, `;`, `(`, `)`, `floatLit`, `not`, `void`, `.`, `*`, `/`, `and`, `isa`, `==`, `>=`, `>`, `<=`, `<`, `<>`, `if`, `then`, `else`, `read`, `return`, `while`, `write`, `float`, `int`, `private`, `public`, `function`, `constructor`, `implementation`, `local`, `=>`

C. S - Starting Symbol

`START`

D. R - Production Rules

```
<START>               ::= <prog>
<prog>                ::= {{<classOrImplOrFunc>}}
<classOrImplOrFunc>   ::= <classDecl> | <implDef> | <funcDef>
<classDecl>           ::= 'class' 'id' [['isa' 'id' {{',' 'id'}}]]
                          '{' {{<visibility> <memberDecl>}} '}' ';'
<implDef>             ::= 'implementation' 'id'
                          '{' {{<funcDef>}} '}'
<funcDef>             ::= <funcHead> <funcBody>
<visibility>          ::= 'public' | 'private'
<memberDecl>          ::= <funcDecl> | <attributeDecl>
<funcDecl>            ::= <funcHead> ';'
<funcHead>            ::= 'function' 'id' '(' <fParams> ')'
                          '=>' <returnType>
                       |  'constructor' '(' <fParams> ')'
<funcBody>            ::= '{' {{<localVarDeclOrStat>}} '}'
<localVarDeclOrStat> ::= <localVarDecl> | <statement>
<attributeDecl>       ::= 'attribute' <varDecl>
<localVarDecl>        ::= 'local' <varDecl>
<varDecl>             ::= 'id' ':' <type> {{<arraySize>}} ';'
<statement>           ::= <assignStat> ';'
                       |  'if' '(' <relExpr> ')' 'then'
                          <statBlock> 'else' <statBlock> ';'
                       |  'while' '(' <relExpr> ')'
                          <statBlock> ';'
                       |  'read' '(' <variable> ')' ';'
                       |  'write' '(' <expr> ')' ';'
                       |  'return' '(' <expr> ')' ';'
                       |  <functionCall> ';'
<assignStat>          ::= <variable> <assignOp> <expr>
<statBlock>           ::= '{' {{<statement>}} '}'
                       |  <statement> | EPSILON
<expr>                ::= <arithExpr> | <relExpr>
<relExpr>             ::= <arithExpr> <relOp> <arithExpr>
<arithExpr>           ::= <arithExpr> <addOp> <term> | <term>
<sign>                ::= '+' | '-'
<term>                ::= <term> <multOp> <factor> | <factor>
<factor>              ::= <variable>
                       |  <functionCall>
                       |  'intLit' | 'floatLit'
                       |  '(' <arithExpr> ')'
                       |  'not' <factor>
                       |  <sign> <factor>
<variable>            ::= {{<idnest>}} 'id' {{<indice>}}
<functionCall>        ::= {{<idnest>}} 'id' '(' <aParams> ')'
<idnest>              ::= <idOrSelf> {{<indice>}} '.'
                       |  <idOrSelf> '(' <aParams> ')' '.'
<indice>              ::= '[' <arithExpr> ']'
<arraySize>           ::= '[' 'intNum' ']' | '[' ']'
<type>                ::= 'int' | 'float' | 'id'
<returnType>          ::= <type> | 'void'
<fParams>             ::= 'id' ':' <type> {{<arraySize>}}
                          {{<fParamsTail>}} | EPSILON
<aParams>             ::= <expr> {{<aParamsTail>}} | EPSILON
<fParamsTail>         ::= ',' 'id' ':' <type> {{<arraySize>}}
<aParamsTail>         ::= ',' <expr>
<assignOp>            ::= ':='
<relOp>               ::= '==' | '<>' | '<' | '>' | '<=' | '>='
<addOp>               ::= '+' | '-' | 'or'
<multOp>              ::= '*' | '/' | 'and'
<idOrSelf>            ::= 'id' | 'self'
```

E. Notes

- Terminals in single quotes (`'token'`), non-terminals in angle brackets (`<nonterm>`)
- `EPSILON` represents empty phrase
- Repetition: `{{phrase}}` for zero or more occurrences
- Optionality: `[[phrase]]` for zero or one occurrence

III. LL(1) Grammar

As mentioned earlier, the given grammar is transformed into an LL(1) non-ambiguous grammar using `grammartool.jar` and `UCalgary` tools. This transformation produces a grammar that can be used in a predictive parser implementation. The resulting grammar is as follows:

```
START               ::= PROG

ADDOP               ::= plus | minus | or

APARAMS             ::= EXPR REPTAPARAMS1 | ϵ
APARAMSTAIL         ::= comma EXPR

ARITHEXPR           ::= TERM RIGHTRECARITHEXPR

ARRAYSIZE           ::= lsqbr ARRAYSIZE2
ARRAYSIZE2          ::= intlit rsqbr | rsqbr

ATTRIBUTEDECL       ::= attribute VARDECL

CLASSDECL           ::= class id OPTCLASSDECL2 lcurbr
                       REPTCLASSDECL4 rcurbr semi

CLASSORIMPLORFUNC   ::= CLASSDECL | IMPLDEF | FUNCDEF

EXPR                ::= ARITHEXPR EXPR2
EXPR2               ::= RELOP ARITHEXPR | ϵ

FACTOR              ::= IDORSELF FACTOR2 REPTVARIABLEORFUNCTIONCALL
                     | intlit
                     | floatlit
                     | lpar ARITHEXPR rpar
                     | not FACTOR
                     | SIGN FACTOR

FACTOR2             ::= lpar APARAMS rpar | REPTIDNEST1

FPARAMS             ::= id colon TYPE REPTFPARAMS3 REPTFPARAMS4 | ϵ
FPARAMSTAIL         ::= comma id colon TYPE REPTFPARAMSTAIL4

FUNCBODY            ::= lcurbr REPTFUNCBODY1 rcurbr
FUNCDECL            ::= FUNCHEAD semi
FUNCDEF             ::= FUNCHEAD FUNCBODY
FUNCHEAD            ::= function id lpar FPARAMS rpar arrow RETURNTYPE
                     | constructor lpar FPARAMS rpar

IDNEST              ::= dot id IDNEST2
IDNEST2             ::= lpar APARAMS rpar | REPTIDNEST1
IDORSELF            ::= id | self

IMPLDEF             ::= implementation id lcurbr REPTIMPLDEF3 rcurbr
INDICE              ::= lsqbr ARITHEXPR rsqbr

LOCALVARDECL        ::= local VARDECL
LOCALVARDECLORSTAT  ::= LOCALVARDECL | STATEMENT
MEMBERDECL          ::= FUNCDECL | ATTRIBUTEDECL
MULTOP              ::= mult | div | and

OPTCLASSDECL2       ::= isa id REPTOPTCLASSDECL22 | ϵ
PROG                ::= REPTPROG0

RELEXPR             ::= ARITHEXPR RELOP ARITHEXPR
RELOP               ::= eq | neq | lt | gt | leq | geq

RETURNTYPE          ::= TYPE | void

RIGHTRECARITHEXPR   ::=  ADDOP TERM RIGHTRECARITHEXPR | ϵ
RIGHTRECTERM        ::=  MULTOP FACTOR RIGHTRECTERM | ϵ

SIGN                ::= plus | minus

STATBLOCK           ::= lcurbr REPTSTATBLOCK1 rcurbr
                     | STATEMENT
                     | ϵ

STATEMENT           ::= IDORSELF SELECTORLIST0 STATEMENTSUFFIX0
                     | if lpar RELEXPR rpar then STATBLOCK
                       else STATBLOCK semi
                     | while lpar RELEXPR rpar STATBLOCK semi
                     | read lpar VARIABLE rpar semi
                     | write lpar EXPR rpar semi
                     | return lpar EXPR rpar semi

SELECTORLIST0       ::= TYPESELECTOR SELECTORLIST0 | ϵ
TYPESELECTOR        ::= lsqbr ARITHEXPR rsqbr | dot id
```

```
STATEMENTSUFFIX0    ::= lpar APARAMS rpar semi
                     | assign EXPR semi

TERM                ::= FACTOR RIGHTRECTERM
TYPE                ::= int | float | id
VARDECL             ::= id colon TYPE REPTVARDECL3 semi

VARIABLE            ::= IDORSELF VARIABLE2
VARIABLE2           ::= REPTIDNEST1 REPTVARIABLE
                     | lpar APARAMS rpar VARIDNEST

VISIBILITY          ::= public | private

REPTVARIABLEORFUNCTIONCALL ::= IDNEST REPTVARIABLEORFUNCTIONCALL | ϵ
REPTAPARAMS1        ::= APARAMSTAIL REPTAPARAMS1 | ϵ
REPTCLASSDECL4      ::= VISIBILITY MEMBERDECL REPTCLASSDECL4 | ϵ
REPTFPARAMS3        ::= ARRAYSIZE REPTFPARAMS3 | ϵ
REPTFPARAMS4        ::= FPARAMSTAIL REPTFPARAMS4 | ϵ
REPTFPARAMSTAIL4    ::= ARRAYSIZE REPTFPARAMSTAIL4 | ϵ
REPTFUNCBODY1       ::= LOCALVARDECLORSTAT REPTFUNCBODY1 | ϵ
REPTIDNEST1         ::= INDICE REPTIDNEST1 | ϵ
REPTIMPLDEF3        ::= FUNCDEF REPTIMPLDEF3 | ϵ
REPTOPTCLASSDECL22  ::= comma id REPTOPTCLASSDECL22 | ϵ
REPTPROG0           ::= CLASSORIMPLORFUNC REPTPROG0 | ϵ
REPTSTATBLOCK1      ::= STATEMENT REPTSTATBLOCK1 | ϵ
REPTVARDECL3        ::= ARRAYSIZE REPTVARDECL3 | ϵ
REPTVARIABLE        ::= VARIDNEST REPTVARIABLE | ϵ

VARIDNEST           ::= dot id VARIDNEST2
VARIDNEST2          ::= lpar APARAMS rpar VARIDNEST | REPTIDNEST1
```

IV. FIRST AND FOLLOW SETS

The FIRST and FOLLOW sets are generated using the UCalgary tool. These sets are all listed in Table I.

V. TABLE-DRIVEN PREDICTIVE PARSER DESIGN

This section details the architecture and implementation of a table-driven predictive parser designed for syntax analysis. The parser leverages a parsing table derived from context-free grammar rules and employs a stack-based algorithm to validate input tokens.

A. Parsing Table Construction

The parsing table is loaded from a CSV file and managed by the `ParsingTable` class. Key features include:

- **CSV Parsing**: The table is initialized by reading terminals (columns) and non-terminals (rows) from the CSV. Productions are stored in a nested `unordered_map` for O(1) lookups.
- **FIRST and FOLLOW Sets**: These are computed iteratively:
  - `computeFirstSets` checks if productions start with terminals or non-terminals, propagating FIRST sets accordingly.
  - `computeFollowSets` uses the FIRST sets of subsequent symbols and propagates FOLLOW sets through productions.
- **Query Methods**: Methods like `getProduction`, `isInFirst`, and `isInFollow` enable direct table lookups during parsing.

B. Parsing Algorithm

The `Parser` class implements the core parsing logic:

- **Initialization**: The stack is initialized with the start symbol (`START`) and an end marker ($). Tokens are fetched via a `Scanner` object.
- **Main Loop**:
  1) If the stack top is a terminal, it is matched against the lookahead token.
  2) For non-terminals, the corresponding production is fetched from the parsing table. If found, the right-hand side (RHS) symbols are pushed onto the stack in reverse order (via `inverseRHSMultiplePush`).
  3) Epsilon productions (`&epsilon`) are skipped during stack pushes as they are empty characters.
- **Derivation Tracking**: The current derivation string is updated by replacing non-terminals with their production RHS.

C. Error Handling and Recovery

The parser employs panic-mode recovery:

- **Token Skipping**: If no valid production exists for a non-terminal and lookahead, tokens are skipped until a valid symbol in `FIRST` or `FOLLOW` sets is found.
- **Follow Set Checks**: If the lookahead is in the `FOLLOW` set of the current non-terminal, the stack is popped to resume parsing.
- Errors are logged with line numbers and unexpected token details.

D. Integration and Execution

The `ParseDriver` class serves as the entry point:

- It accepts command-line arguments for input files and custom parsing tables.
- For each input file, a `Parser` instance is created, and the `parse()` method is invoked.
- Derivations and syntax errors are written to `.outderivation` and `.outsyntaxerrors` files, respectively.

E. Key Implementation Details

- **Stack Management**: The `parseStack` ensures deterministic transitions by enforcing reverse order insertion of RHS symbols.
- **Efficiency**: The use of hash maps for the parsing table and sets for `FIRST`/`FOLLOW` ensures efficient symbol lookups.
- **Modularity**: The separation of `ParsingTable`, `Parser`, and `Scanner` adheres to modular design principles.

This implementation provides a robust framework for syntax analysis, combining theoretical parsing concepts with practical error recovery mechanisms.

TABLE I

TABLE OF FIRST AND FOLLOW SETS FOR EACH NON-TERMINAL

Nonterminal	FIRST Set	FOLLOW Set
START	class implementation function constructor	
ARRAYSIZE2	intlit rsqbr	semi lsqbr rpar comma
CLASSDECL	class	class implementation function constructor
EXPR2	eq neq lt gt leq geq	comma rpar semi
FACTOR2	lpar lsqbr	mult div and dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
REPTVARIABLEORFUNCTIONCALL	dot	mult div and eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
FUNCBODY	lcurbr	class implementation function constructor rcurbr
FUNCHEAD	function constructor	semi lcurbr
FPARAMS	id	rpar
IDNEST	dot	mult div and dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
IDNEST2	lpar lsqbr	mult div and dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
IMPLDEF	implementation	class implementation function constructor
LOCALVARDECL	local	local if while read write return id self rcurbr
FUNCDECL	function constructor	public private rcurbr
ATTRIBUTEDECL	attribute	public private rcurbr
OPTCLASSDECL2	isa	lcurbr
PROG	class implementation function constructor	
RELOP	eq neq lt gt leq geq	intlit floatlit lpar not id self plus minus
APARAMSTAIL	comma	comma rpar
REPTAPARAMS1	comma	rpar
MEMBERDECL	attribute function constructor	public private rcurbr
REPTCLASSDECL4	public private	rcurbr
REPTFPARAMS3	lsqbr	rpar comma
FPARAMSTAIL	comma	comma rpar
REPTFPARAMS4	comma	rpar
REPTFPARAMSTAIL4	lsqbr	comma rpar
LOCALVARDECLORSTAT	local if while read write return id self	local if while read write return id self rcurbr
REPTFUNCBODY1	local if while read write return id self	rcurbr
INDICE	lsqbr	mult div and lsqbr dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
FUNCDEF	function constructor	class implementation function constructor rcurbr
REPTIMPLDEF3	function constructor	rcurbr
REPTOPTCLASSDECL22	comma	lcurbr
CLASSORIMPLORFUNC	class implementation function constructor	class implementation function constructor
REPTPROG0	class implementation function constructor	
ARRAYSIZE	lsqbr	semi lsqbr rpar comma
RETURNTYPE	void int float id	semi lcurbr
ADDOP	plus minus or	intlit floatlit lpar not id self plus minus
RIGHTRECARITHEXPR	plus minus or	eq neq lt gt leq geq rsqbr comma rpar semi
MULTOP	mult div and	intlit floatlit lpar not id self plus minus
SIGN	plus minus	intlit floatlit lpar not id self plus minus
REPTSTATBLOCK1	if while read write return id self	rcurbr
STATEMENT	if while read write return id self	else semi local if while read write return id self rcurbr
RELEXPR	intlit floatlit lpar not id self plus minus	rpar
STATBLOCK	lcurbr if while read write return id self	else semi
SELECTORLIST0	lsqbr dot	lpar assign
TYPESELECTOR	lsqbr dot	lsqbr dot lpar assign
ARITHEXPR	intlit floatlit lpar not id self plus minus	eq neq lt gt leq geq rsqbr comma rpar semi
STATEMENTSUFFIX0	lpar assign	else semi local if while read write return id self rcurbr
EXPR	intlit floatlit lpar not id self plus minus	comma rpar semi
TERM	intlit floatlit lpar not id self plus minus	eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
FACTOR	intlit floatlit lpar not id self plus minus	mult div and eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
RIGHTRECTERM	mult div and	eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
VARDECL	id	local if while read write return id self public private rcurbr
TYPE	int float id	rpar lcurbr comma lsqbr semi
REPTVARDECL3	lsqbr	semi
VARIABLE	id self	rpar
IDORSELF	id self	mult div and assign lpar lsqbr dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
VARIABLE2	lpar lsqbr dot	rpar
REPTVARIABLE	dot	rpar
VARIDNEST2	lpar lsqbr	dot rpar
APARAMS	intlit floatlit lpar not id self plus minus	rpar
VARIDNEST	dot	dot rpar
REPTIDNEST1	lsqbr	mult div and dot eq neq lt gt leq geq rsqbr plus minus or comma rpar semi
VISIBILITY	public private	attribute function constructor