Retro Basic

Phirasit Charoenchitseriwong | 5931043321
Programming Language Principles 2110316 | First Semester 2018

This report is a summary report of the project Retro Basic, a Microsoft Basic compiler. The details of this project can be found here: https://www.cp.eng.chula.ac.th/~piak/teaching/prolang/2018/retro-basic.htm. The implementation of the actual compiler can be found here: https://www.github.com/phirasit/RetroBasic.

0. Introduction

The original Basic compiler is a compiler that takes only 4KB memory so there would be more memory left to run programs. Unfortunately, this number is not easily achievable in today's computer architecture. Despite that, the grammar of the Basic language can be replicated without much complication. The basic grammar of Microsoft Basic language in this compiler is

```
pgm := line pgm | EOF
line := line_num stmt
stmt := asgmnt | if | print | goto | stop
asgmnt := id = exp
exp := term + term | term - term
term := id | const
if := IF cond line_num
cond := term < term | term = term
print := PRINT id
goto := GOTO line_num
stop := STOP
```

However, this grammar does not qualify as an LL-1 grammar not to mention the fact that it fails to parse the sample inputs. There have to be some adjustments to make the parsing process more efficient. The revised grammar is

```
pgm := line pgm | EOF
line := line_num stmt
stmt := asgmnt | if | print | goto | stop
asgmnt := id = exp
* exp := term exp2
* exp2 := + term | - term | empty
term := id | const
if := IF cond line_num
* cond := term cond2
* cond2 := < term | = term
print := PRINT id
goto := GOTO line_num
stop := STOP
```

This grammar is the grammar that is in the implementation. The compiler is implemented in Haskell and can be found in the Github link. The output of the compiler will be a B-code which is an intermediate code. The advantage of immediate code is that it is easy to optimize and convert to machine code. Unfortunately, the process after the compilation is not covered in this report.

To demo the program, one should clone the source code from Github and build the program from build file (Makefile). All the installation and running detail can be found in "README.md" in the source folder. There might be some complication compiling the Haskell source code but that should not be too hard. In the folder there is a lister folder. Lister is a tool to decompile the B-code back to the Basic language. It is used to test the compiler.

1. Scanner

This part is dedicated to how to scan and tokenize text file into labels predefined in the Basic grammar. The implementation of this part is in the file named “Scanner.hs”. In this part, the scanner will separate the input into the following terminal tokens:

```
ID := A..Z
** LINE_NUM := 0 .. 1000
** CONST := 0 .. 100
IF := “IF”
PRINT := “PRINT”
GOTO := “GOTO”
STOP := “STOP”
+ := “+”
- := “-”
< := “<”
= := “=”
EOF := end-of-file
```

**** NOTE: LINE_NUM, CONST** cannot be explicitly implied during this step. All number will be considered as **CONST** during this stage of compilation. Type and constraint determination will be done later in the parsing stage.

The scanner can be implemented using DFA (Deterministic Finite Automata) because every leading token is all distinct. It should be noted that many implementations (including this one) may choose other methods instead for better performance. After this process, a text file is now converted into a list of defined tokens ready for parsing.

2. Parser

This part is how compiler parse the tokens from the first step into an **abstract syntax tree (AST)**. AST can be easily converted into immediate code which can help with the optimization process. As mention in the introduction, this grammar is qualified as an LL-1 grammar which means only one token needs to be looked up in order to determine which rules to use. The implementation of this part of the code is in the file “Parser.hs”.

To parse the tokens, first, a parsing table has to be established. This part requires the information regarding first sets and follow sets listing below.

Table 2.1: First-set and Follow-set

No.	Grammar rule	First set	Follow set
1	`pgm := line pgm`	`LINE_NUM`	`EOF`
2	`pgm := EMPTY`		
3	`line := LINE_NUM stmt`	`LINE_NUM`	`LINE_NUM` `EOF`

4	`stmt := asgmnt`	`ID`
		`IF`
		`PRINT`
		`GOTO`
		`STOP`
			`LINE_NUM`
			`EOF`

5	`stmt := if`		
6	`stmt := print`		
7	`stmt := goto`		
8	`stmt := stop`		
9	`asgmnt := ID = exp`	`ID`	`LINE_NUM` `EOF`
10	`exp := term exp2`	`ID` `CONST`	`LINE_NUM` `EOF`
11	`exp2 := + term`	`+`	`LINE_NUM`

12	exp2 := - term	- EMPTY	EOF
13	exp2 := EMPTY		
14	term := ID	ID CONST	+ - < = ID CONST LINE_NUM EOF
15	term := CONST		
16	if := IF cond LINE_NUM	IF	LINE_NUM EOF
17	cond := term cond2	ID CONST	LINE_NUM
18	cond2 := < term	< =	LINE_NUM
19	cond2 := = term		
20	print := PRINT ID	PRINT	LINE_NUM EOF
21	goto := GOTO LINE_NUM	GOTO	LINE_NUM EOF
22	stop := STOP	STOP	LINE_NUM EOF

Table 2.2: Parsing table

Non-terminal	ID	LINE_NUM	CONST	IF	PRINT	GOTO	STOP	+	-	<	=	EOF
pgm		1										2
line		3										
stmt	4			5	6	7	8					
asgmnt	9											
exp	10		10									
exp2		13						11	12			13
term	14		15									
if				16								
cond	17		17									
cond2										18	19	
print					20							
goto						21						
stop							22					

In the parsing process, a stack is used to keep track of all remaining non-terminals. Each non-terminal on the top of the stack will be derived according to the parsing table and the following token. Each new non-terminals that come from derivation will go to the stack in the reverse order. If no rule can be applied, the compilation will stop with a compilation error message otherwise a node in the AST is created.

3. Implementation

As mention before, all the code is written in Haskell, a functional language. In functional languages, there are features called “lazy propagation” and “pattern matching” which will be heavily used in this compiler.

The primary files in the RetroBasic project are

- Scanner.hs Convert input text into tokens
- Parser.hs Parse tokens into AST according to the defined grammars
- Emitter.hs Convert AST into B-code

Each file contains the functions for its intended purposes and is included in “main.hs”. “main.hs” contains I/O wrapper to make it easier to use. The implementation of all other files in the project is not important and can be skipped.

Scanner.hs

```
module Scanner where

import Data.Char (isDigit, isSpace, isUpper)

data Token = Const Int
  | Id Char
  | If
  | Goto
  | Print
  | Stop
  | Plus
  | Minus
  | Less
  | Equal
  deriving (Show)

trim :: String -> String
trim = f . f
   where f = reverse . dropWhile isSpace

tokenize :: String -> Token
tokenize "IF" = If
tokenize "GOTO" = Goto
tokenize "PRINT" = Print
tokenize "STOP" = Stop
tokenize "+" = Plus
tokenize "-" = Minus
tokenize "<" = Less
tokenize "=" = Equal
tokenize w@(x:xs)
  | isDigit x = Const $ (read :: String -> Int) w
  | isUpper x && null xs = Id x
  | otherwise = error $ "Scanner: Token Unrecognize " ++ (show x) ++ " " ++ (show xs)

tokenize t = error $ "Scanner: Token Unrecognize " ++ (show t)

scan :: String -> [Token]
scan w = map tokenize $ filter (/="") $ map trim $ words w
```

Tokenizing can be done by using DFA. In this implementation, however, the “scan” function will be used. This function will split the string w into a list of strings, trim the strings, and filter out the empty string (can be seen in the last line). Each string then will be converted into “Token” using “tokenize” function.

the “tokenize” function will try to match each string into a token. Each reserved word in this language will be matched first according to the matching rule of the Haskell language. If the label is not a reserved word, the type of word can only be **ID** or **CONST** which will be determined by the type its first letter.

If the label fails to match any case, the program will fail with an error “Token Unrecognize” that can be seen in the code above. If no error is reported, the function will return a list of datatype “Token” which will be used in the following process.

Parser.hs

```
...

createAST Terminal → [Token] → (AST, [Token])

-- pgm := line pgm | EOF
createAST PGM [] = (Null, [])
createAST PGM tokens@((Scanner.Const _):_) = (Parser.Pgm ast1 ast2, tokens'')
 where
  (ast1, tokens') = createAST LINE tokens
  (ast2, tokens'') = createAST PGM tokens'
createAST PGM tokens = error $ "Parser: Invalid PGM " ++ (show tokens)

-- line := line_num stmt
createAST LINE (lineNum@(Scanner.Const _):tokens)
 | isLineNum lineNum = (Parser.Line lineNum ast, tokens')
 | otherwise = error $ "Parser: Invalid LINE " ++ (show lineNum)
 where (ast, tokens') = createAST STMT tokens
createAST LINE tokens = error $ "Parser: Invalid LINE " ++ (show tokens)

...
```

To parse the token the function “createAST” is used. Only two tokens are shown in this report for convenience. This function will take a terminal symbol and an input stream. Since **PGM** contains two rules, in this case, there will be two patterns to match (including 1 error matching for compilation error message). The return value of the function will be a tuple of a generated AST and the remaining of the stream. The recursions will serve as a stack like the one in the parsing procedure.

Similarly, **LINE only** contains one rule which requires a leading token to be **LINE_NUM**. If the leading token is not **LINE_NUM** the whole parsing sequence is terminated with an error. Other rules are implemented in a similar way and can be found in the file.

Emitter.hs

```
Module Emitter (toBCode) where

...

toBCodeInline :: AST → [Int]

toBCodeInline (Parser.Line (Scanner.Const n) ast1) = lineCode : n : (toBCodeInline ast1)
toBCodeInline (Parser.Line token _) = error $ "Compiler: Invalid Line " ++ (show token)

toBCodeInline (Parser.Stmt ast1) = toBCodeInline ast1

toBCodeInline (Parser.Asgmnt (Scanner.Id c) Scanner.Equal ast1) = (showIdCode c) ++ (showOpCode Scanner.Equal) ++ (toBCodeInline ast1)
toBCodeInline (Parser.Asgmnt _ _ _) = error "Compiler: Invalid Asgmnt"

toBCodeInline (Parser.Exp1 ast1 ast2) = (toBCodeInline ast1) ++ (toBCodeInline ast2)
toBCodeInline (Parser.Exp21) =  []
toBCodeInline (Parser.Exp22 op ast1) = (showOpCode op) ++ (toBCodeInline ast1)

toBCodeInline (Parser.Term (Scanner.Id c)) = showIdCode c
toBCodeInline (Parser.Term (Scanner.Const n)) = showConstCode n
toBCodeInline (Parser.Term token) = error $ "Compiler: Invalid Term " ++ (show token)

toBCodeInline (Parser.If ast1 (Scanner.Const n)) = showIfCode ++ (toBCodeInline ast1) ++ (showGotoCode n)
toBCodeInline (Parser.If _ token) = error $ "Compiler: Invalid If " ++ (show token)

toBCodeInline (Parser.Cond1 ast1 ast2) = (toBCodeInline ast1) ++ (toBCodeInline ast2)
toBCodeInline (Parser.Cond2 op ast) = (showOpCode op) ++ (toBCodeInline ast)

toBCodeInline (Parser.Print (Scanner.Id c)) = showPrintCode ++ (showIdCode c)
toBCodeInline (Parser.Print token) = error $ "Compiler: Invalid Print " ++ (show token)

toBCodeInline (Parser.Goto (Scanner.Const n)) = showGotoCode n
toBCodeInline (Parser.Goto token) = error $ "Compiler: Invalid Goto " ++ (show token)

toBCodeInline Parser.Stop = showStopCode

toBCodeInline ast = error $ "Compiler: Invalid AST " ++ (show ast)

...
```

The final part is to convert AST into B-code. All of this part is done inside “Compiler.hs”. The function “toBCodeInline” takes an AST and convert it into a list of B-Codes. Each line is one of the patterns of ASTs. This function also requires some helper functions which are predefined in the lines before this part of the code.