Retro Basic Compiler

Phon Chitnelawong

1 Retro Basic Programming Language

1.1 Grammar Rules

The provided grammar cannot be parsed with LL1 parser. I modified the grammar to satisfy LL1 parsing constraints. The results are as follows:

[0]	pgm	→	line pgm
[1]	pgm	→	EOF
[2]	line	→	line_num stmt
[3]	stmt	→	asgmnt
[4]	stmt	→	if
[5]	stmt	→	print
[6]	stmt	→	goto
[7]	stmt	→	stop
[8]	asgmnt	→	id = exp
[9]	exp	→	term **exp2**
[10]	**exp2**	→	+ term
[11]	**exp2**	→	- term
[12]	term	→	id
[13]	term	→	const
[14]	if	→	IF cond line_num
[15]	cond	→	term **cond2**
[16]	**cond2**	→	< term
[17]	**cond2**	→	= term
[18]	print	→	PRINT id
[19]	goto	→	GOTO line_num
[20]	stop	→	STOP
[21]	exp2	→	λ

Notice that **exp2** and **cond2** non-terminal symbols are added to the rules.

1.2 First Set & Follow Set

The sets are computed as:

Non-Terminal Symbol	First Set	Follow Set
pgm	EOF, line_num	EOF
line	line_num	EOF, line_num
stmt	id, PRINT, STOP, IF, GOTO	EOF, line_num
asgmnt	id	EOF, line_num
exp	id, const	EOF, line_num
exp2	$+, -, \lambda$	EOF, line_num
term	id, const	<, =, +, −, EOF, line_num
if	IF	EOF, line_num
cond	id, const	line_num
cond2	<, =	line_num
print	PRINT	EOF, line_num
goto	GOTO	EOF, line_num
stop	STOP	EOF, line_num

1.3 Parsing Table

Using the rules, the first set and the follow set above the parsing table can then be built as follows:

	id	**const**	**line_num**	**GOTO**	**STOP**	**IF**	**PRINT**	**+**	**-**	**<**	**=**	**EOF**
pgm			0									1
line			2									
stmt	3			6	7	4	5					
asgmnt	8											
exp	9	9	9									9
exp2								10	11			
term	12	13										
if						14						
cond	15	15										
cond2										16	17	
print							18					
goto				19								
stop					20							

2 The Compiler

My Retro Basic compiler is written in Python version 3.

2.1 The Scanner

By using Python, implementing a scanner is very easy. I just needed to read each line of the source code, strip new line character and split symbols into a list.

```
while len(line) != 0:
    line = src_file.readline()
    tokens = line.strip().split()
    # parse the tokens
    ...
```

2.2 The Parser

The parser is a little bit complex. First, I needed to build a LL1 parser stack (like one in the slides). In the stack, there are terminals and non-terminals. Terminals are denoted as strings, and non-terminals are denoted as number (enumeration). The stack is initialized with an **EOF** and a non-terminal **pgm** as starting symbol (The item on top of the stack is **pgm**.).

The parsing process is applied for each token list in one line. The parser iterates through all tokens in the list. First, it determines the token's type (e.g. id, const, etc.). If the token's type cannot be determined, the compiler throws an error and exits, otherwise the parser continues to the next step.

After the token's type has been determined, the parser check whether the symbol on top of the stack is a terminal or not. If it is a terminal, the parser tries to match the symbol with current token, if it matches, the stack is then popped and the parser proceeds to process the next token (Conversion to bcode happens in this step.), otherwise it throws an error indicating that a symbol on top of the stack is expected. But if the symbol on top of the stack is a non-terminal symbol, the parser looks for a grammar rule from the parsing table with respect to the current token's type. The parser looks for the grammar such that the starting symbol matches the one on top of the stack, if it matches, the symbol on top of the stack is substituted with the production from right to left. If none of the rules are matched, the parser checks if there exists a production corresponding to the symbol on top of the stack, which can produce an empty string (λ). If it does exists, that symbol is then gracefully popped out of the stack. The compiler throws an error and exits if nothing can be matched, pointing that the source code is written in wrong syntax.

The compiler repeats the steps above until it reaches the end of the source file, when the bcodes are writen to output and the compiler exits.

Here is a pseudo-code of the parser (It is very similar to Python, though.):

```
ll_stack = []
...
# scanner code above
...
token_index = 0
while token_index < len(tokens):
    token = tokens[token_index]
    term_types = terminal_type(token)
    top = ll_stack[-1]
    if len(term_types) == 0:
        UNRECOGNIZED SYMBOL ERROR
        exit()
    type = term_types[0]
    if is_terminal(top):
        if type == top:
            write_bcode(type, token)
            token_index += 1
            ll_stack.pop()
        else:
            EXPECTED TOKEN ERROR
            exit()
    else:
        accept = false
        for rule in parsing_table[type]:
            if starting_symbol(rule) == top:
                ll_stack.pop()
                ll_stack.push(reverse(production_of(rule)))
                accept = true
                break
        if not accept:
            for rule in rules:
                if starting_symbol(rule) == top:
                    if λ in production(rule):
                        accept = true
                        ll_stack.pop()
                        break
        if not accept:
            UNEXPECTED TOKEN ERROR
            exit()
```

3 Actual Code

The actual working code for my compiler has been hosted at `https://github.com/phonxvzf/retroc`.