TINY Language
Lexical Parser
Experimental Report

田野
2017329621125
Software Environment: Max+flex

1. Experimental Purpose

Build the lexical parser (a scanner) of TINY language, using the Lex tool of third party. The experimental result Scanner can receive the sample program of Tiny language, and output is a sequence of tokes that are defined by regular expression.

2. Experimental Content

2.1 Description of TINY language

The TINT language has three types of token those are reserved word, specific symbol and others(identifier and integer).

Reserved Word	Specific symbol	others
if	+	Identifier （one or more word）
then	-	
else	*	
end	/	
repeat	=	Integer （one or more integer）
until	<	
read	>	
write	(	
	)	
	;	
	:=	

To recognize each token, we need to write regular expression for each token. For reserved word and specific symbol, the regular expression is much simpler. For example, the regular expression for reserved word if is “if”. But regular expression for others is much more complicated. For identifier, according to the definition of identifier, the regular expression

is ([_a-zA-Z][_a-zA-Z0-9]*). For integer, the regular expression is ([0-9]+).

2.2 Experimental requirement

- Read context from extern file
- Remove whitespace characters (Spaces, carriage returns, tabs)
- Find token in the file then print the token with token type
- Find and positioning error

2.3 Introduction for flex

The complete format of flex program is:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

Paragraph 1 of the input file % {and} % for statement (Declarations), the declarations are C code, the code will be the same copy to lex.yy.c file, generally we will state some global variables and functions here, so that we can use these variables and functions later.

Paragraph 2 between %}and%% is definitions. We can definite some regular expression here so that we can use those regular expression in Rules.

Paragraph 3 between %% and %% is rules. Each line in this file is a rule. Each rule consists of a pattern and an action. The pattern is in the front, expressed by regular expressions, and the event is in the back, that is, C code. Every time a pattern is matched, the following C code is executed.

Paragraph 4 is User subroutines that is C code. The contents of this paragraph will be copied to the end of the yylex. C file as is, and the functions declared in paragraph 1 are generally defined here. In the four paragraphs above, except for the Rules section, all three are optional.

3. Experimental Realization

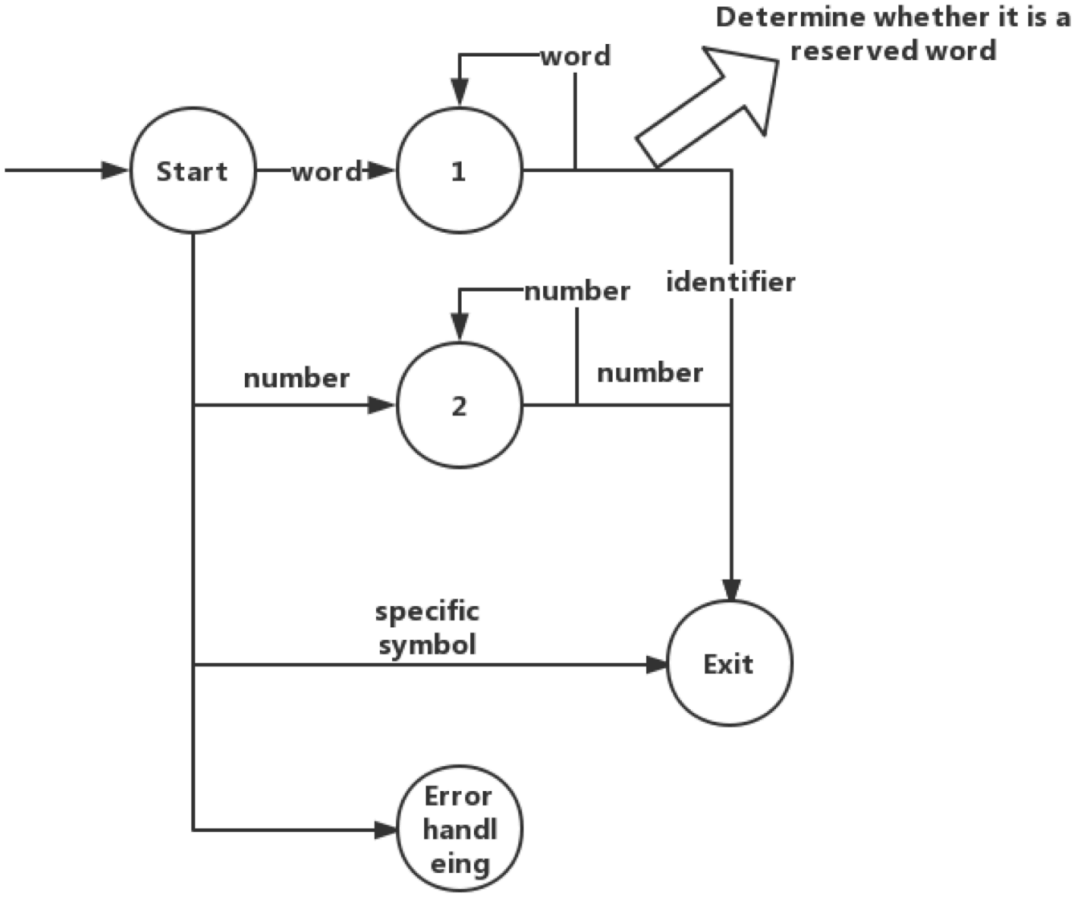
3.1 Specific design of lexical parser

The type of the binary form of the token is defined by the enumeration method; the reserved word and the special character are each of a kind, the identifier itself is a class, the number is a class; the attribute of the word is the string value represented.

The specific function implementation of lexical analysis is a function. Each call analyzes the remaining string to get a word or token to identify its kind, collects the symbol string attribute of the token, and when the word is recognized, returns the symbol in the form of a return value. The type of the attribute, while in the form of a program variable, provides the attribute value that currently identifies the token. This generates a syntax tree in conjunction with the tokens and their attributes required for the analysis of the parser.

The lexical composition of the identifier and the reserved word are the same. For better implementation, the reserved word of the language is stored in a table, so that the identification of the reserved word can be placed after the identifier, and the identified identifier is used to compare the table. If there is a reserved word in the table, it is a general identifier.

3.2 state transition diagram



4. Experimental Result

4.1 input

we use TINY language demo in the book as input file. The demo is:

```
read x;
if 0 < x then
    fact := 1;
    repeat
        fact := fact * x;
        x := x - 1
    until x = 0;
    write fact
end
```

We feed our scanner the demo, expected the scanner will find out all

token and its type.

4.2 output

The output is:

```
TOKEN-TYPE                TOKEN-VALUE
-----------------------------------------------
Token_Read                  read
T_Identifier              x
Token_Sem                   ;
Token_If                  if
Toker_Integer             0
Token_Le                    <
T_Identifier              x
Token_Then                  then
T_Identifier              fact
Token_War                   :=
Toker_Integer             1
Token_Sem                   ;
Token_Repeat              repeat
T_Identifier              fact
Token_War                   :=
T_Identifier              fact
Token_Mul                   *
T_Identifier              x
Token_Sem                   ;
T_Identifier              x
Token_War                   :=
T_Identifier              x
Token_Minus                 -
Toker_Integer             1
Token_Until               until
T_Identifier              x
```

Token_Eq =
Toker_Integer 0
Token_Sem ;
Token_Write write
T_Identifier fact
Token_End end

As expected, the scanner find all the tokens and their type.

6. Code

mytoken.h

```
#ifndef TOKEN_H
#define TOKEN_H



typedef enum{
   T_If = 256, T_Then, T_Else, T_End, T_Repeat, T_Until, T_Read, T_Write,
   T_Plus, T_Minus, T_Mul, T_Div, T_Eq, T_Le, T_Left, T_Right, T_Sem, T_War,
   T_Integer,  T_Identifier
}TokenType;

static void print_token(int token) {
   static char* token_strs[] = {
      "Token_If", "Token_Then", "Token_Else", "Token_End", "Token_Repeat", "To
ken_Until", "Token_Read",
      "Token_Write", "Token_Plus", "Token_Minus", "Token_Mul", "Token_Div",
      "Token_Eq", "Token_Le", "Token_Left", "Token_Right", "Token_Sem", "Toke
n_War",
       "Toker_Integer", "T_Identifier"
   };

   if (token < 256) {
```

```
        printf("%-20c", token);
    } else {
        printf("%-20s", token_strs[token-256]);
    }
}
#endif
```

scanner.l

```
%{
#include "mytoken.h"
int cur_line_num = 1;
void init_scanner();
void lex_error(char* msg, int line);
%}

/* Definitions, note: \042 is '"' */
INTEGER             ([0-9]+)
IDENTIFIER          ([_a-zA-Z][_a-zA-Z0-9]*)
SINGLE_COMMENT1     ("//"[^\n]*)
SINGLE_COMMENT2     ("#"[^\n]*)

%%

[\n]                { cur_line_num++;                   }
[ \t\r\a]+          { /* ignore all spaces */           }
{SINGLE_COMMENT1}   { /* skip for single line comment */  }
{SINGLE_COMMENT2}   { /* skip for single line commnet */  }



"if"                {return T_If;       }
"then"              {return T_Then;     }
```

```
"else"            {return T_Else;    }
"end"             {return T_End;     }
"repeat"          {return T_Repeat;  }
"until"           {return T_Until;   }
"read"            {return T_Read;    }
"write"           {return T_Write;   }
"+"               {return T_Plus;    }
"-"               {return T_Minus;   }
"*"               {return T_Mul;     }
"/"               {return T_Div;     }
"="               {return T_Eq;      }
"<"               {return T_Le;      }
"("               {return T_Left;    }
")"               {return T_Right;   }
";"               {return T_Sem;     }
":="              {return T_War;     }



{INTEGER}         { return T_Integer;    }
{IDENTIFIER}      { return T_Identifier;     }

<<EOF>>           { return 0; }

.                 { lex_error("Unrecognized character", cur_line_num);       }

%%

int main(int argc, char* argv[]) {
    int token;
    init_scanner();
    while (token = yylex()) {
        print_token(token);
```

```
        puts(yytext);
    }
    return 0;
}

void init_scanner() {
    printf("%-20s%s\n", "TOKEN-TYPE", "TOKEN-VALUE");
    printf("------------------------------------------------\n");
}

void lex_error(char* msg, int line) {
    printf("\nError at line %-3d: %s\n\n", line, msg);
}

int yywrap(void) {
    return 1;
}
```