# The report of course project

### 1. The scanner implementation

This scanner can separate streams from the original code into several tokens which will be used in compiling. There are several kinds of tokens, such as—— operator, id, delimiter, number, bool symbols, key words and so on.

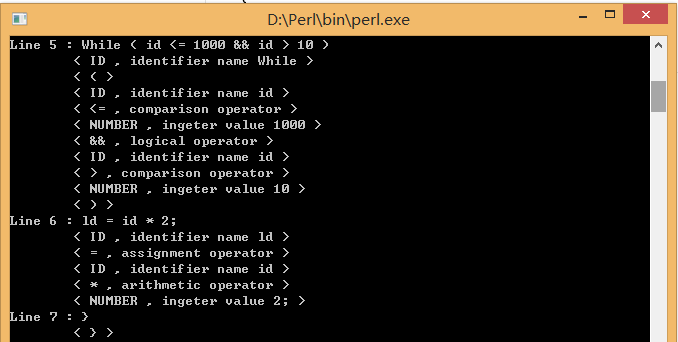
First, the code can create some array for each type of token, such as @id, @key, @op.

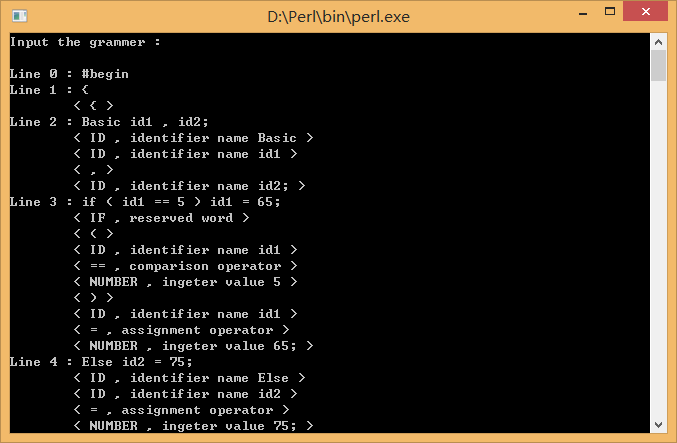
Secondly, using open method to read a sample code file, then use @line=<FILE> to read it in lines and delete the \r of each line using chop(@line)after that, use open method to create a new text file which will save the tokens which has been separated from the original code.

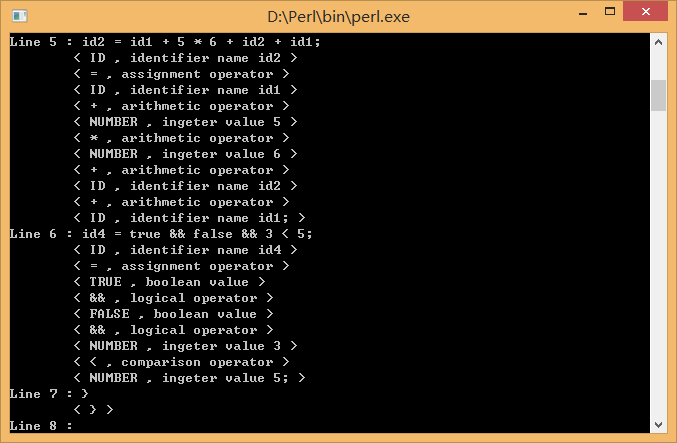
Third, while statement is used to loop each line, in order to compare each char of one line stored an array @char with the kinds of token which have been mentioned above to judge if this char should be added as another token or not. Such as, if the second “=” in original code “==“ should be added as another token.

The flowing diagram is the sample output of scanner part.









### 2: Parse table construction

In order to fill the parsing table, we have to establish what grammar rule the parser should choose if it sees a nonterminal A on the top of its stack and a symbol a on its input stream. It is easy to see that such a rule should be of the form A → w and that the language corresponding to w should have at least one string starting with a. For this purpose we define the First-set of w, written here as Fi(w), as the set of terminals that can be found at the start of some string in w, plus ε if the empty string also belongs to w. Given a grammar with the rules A1 → w1, ..., An → wn, we can compute the Fi(wi) and Fi(Ai) for every rule as follows:

1. initialize every Fi(wi) and Fi(Ai) with the empty set

2. add Fi(wi) to Fi(Ai) for every rule Ai → wi, where Fi is defined as follows:

• Fi(a w' ) = { a } for every terminal a

• Fi(A w' ) = Fi(A) for every nonterminal A with ε not in Fi(A)

• Fi(A w' ) = Fi(A) \ { ε } ∪ Fi(w' ) for every nonterminal A with ε in Fi(A)

• Fi(ε) = { ε }

3. add Fi(wi) to Fi(Ai) for every rule Ai → wi

4. do steps 2 and 3 until all Fi sets stay the same.

Unfortunately, the First-sets are not sufficient to compute the parsing table. This is because a right-hand side w of a rule might ultimately be rewritten to the empty string. So the parser should also use the rule A → w if ε is in Fi(w) and it sees on the input stream a symbol that could follow A. Therefore we also need the Follow-set of A, written as Fo(A) here, which is defined as the set of terminals a such that there is a string of symbols αAaβ that can be derived from the start symbol.

Computing the Follow-sets for the nonterminals in a grammar can be done as follows:

1. initialize every Fo(Ai) with the empty set

2. if there is a rule of the form Aj → wAiw' , then

• if the terminal a is in Fi(w' ), then add a to Fo(Ai)

• if ε is in Fi(w' ), then add Fo(Aj) to Fo(Ai)

• if w' has length 0, then add Fo(Aj) to Fo(Ai)

3. repeat step 2 until all Fo sets stay the same.

Now we can define exactly which rules will be contained where in the parsing table. If T[A, a] denotes the entry in the table for nonterminal A and terminal a, then

T[A,a] contains the rule A → w if and only if

a is in Fi(w) or

ε is in Fi(w) and a is in Fo(A).

If the table contains at most one rule in every one of its cells, then the parser will always know which rule it has to use and can therefore parse strings without backtracking. It is in precisely this case that the grammar is called an LL(1) grammar.

The flowing is the grammar simplify procedure

Original grammar: E --> E a1 | E a2 | ... | E an | b1 | b2 | ... | bn

After refined: E --> b1 E' | b2 E' | ... | bn E'

E'--> a1 E' | a2 E' | ... | an E' | epsilon

find all the left recursion generative production

decls → decl decls | ε

type → basic type'

type' → [ num ] type' | ε

stmts → stmt stmts | ε

loc → id loc'

loc' → [ bool ] loc' | ε

bool → join bool'

bool' → || join bool' | ε

join → equality join'

join' → && equality join' | ε

equality → rel equality'

equality' → == rel equality' | != rel equality' | ε

expr →term expr'

expr' → + term expr' | - term expr' | ε

term → unary term'

term' → \* unary term' | / unary term' | ε

the flowing ins follow set .

sub FollowSet {

# First put $ (the end of input marker) in Follow(S) (S is the start symbol)

my $countChanged = 0;

foreach my $A (@producs) {

my @B = @$A;

my $len = @B;

for ( my $i = $len - 2 ; $i > 1 ; $i-- ) {

if ( is\_member( $B[$i], @VN ) ) {

# If there is a production A → aBb, (where a can be a whole string)

# then everything in FIRST(b) except for ε is placed in FOLLOW(B).

if ( $i + 1 < $len ) {

if ( is\_member( $B[ $i + 1 ], @VN ) ) { #non-terminal

$countChanged += myPush3(

$followSets{ $B[$i] },

$firstSets{ $B[ $i + 1 ] }

);

}

else { #terminal

$countChanged +=

myPush2( $followSets{ $B[$i] }, $B[ $i + 1 ] );

}

}

}

}

if ( $len - 1 > 0 ) {

if ( is\_member( $B[ $len - 1 ], @VN ) ) {

# If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)

$countChanged +=

myPush3( $followSets{ $B[ $len - 1 ] },

$followSets{ $B[0] } );

if ( ( $len - 2 > 0 )

and is\_member( 'ε', @{ $firstSets{ $B[ $len - 1 ] } } ) )

{

# If there is a production A → aBb, where FIRST(b) contains ε,

# then everything in FOLLOW(A) is in FOLLOW(B)

$countChanged += myPush3( $followSets{ $B[ $len - 2 ] },

$followSets{ $B[0] } );

}

}

}

}

return $countChanged;

}

## Flowing is the sample output of the parser





The above diagram is the production of the derivation process

var1 is the production of this time deduction

var2 analysis the stack  
var3 analysis the stack and the stack terminal and non-terminal

var4 is processing token

This is shown in different ways, not only the output of the program, you need to see the source code, but this is not the final table, the final sub-process which T based on these two variables, select generate. The program can no form is not the form, but ranks number one input and output corresponding to the contents of the grid function. So the program do not need to have to be printed as a strict table style

### 3. the derivation starting from *program* to the source code .

 A derivation proceeds in the same order as the execution of the program instructions.

 During a derivation, for each non-terminal expansion that we do, if we create a tree node to represent the non-terminal and add it as a child to the node representing the non-terminal we expanded beforehand, we'll build up a tree of nodes representing the sentence structure.

 A preorder tree traversal on our AST visits nodes in the same order as the derivation, so logically it follows that the traversal follows the order in which instructions are executed. 

 A table-based parsing algorithm utilizes a table that's dynamically generated from the output of the first and follow sets, which can be programmatically generated from a grammar specification.

 Parser generators are implementations of the table-construction algorithms.

sub FollowSet {

# First put $ (the end of input marker) in Follow(S) (S is the start symbol)

my $countChanged = 0;

foreach my $A (@producs) {

my @B = @$A;

my $len = @B;

for ( my $i = $len - 2 ; $i > 1 ; $i-- ) {

if ( is\_member( $B[$i], @VN ) ) {

# If there is a production A → aBb, (where a can be a whole string)

# then everything in FIRST(b) except for ε is placed in FOLLOW(B).

if ( $i + 1 < $len ) {

if ( is\_member( $B[ $i + 1 ], @VN ) ) { #non-terminal

$countChanged += myPush3(

$followSets{ $B[$i] },

$firstSets{ $B[ $i + 1 ] }

);

}

else { #terminal

$countChanged +=

myPush2( $followSets{ $B[$i] }, $B[ $i + 1 ] );

}

}

}

}

if ( $len - 1 > 0 ) {

if ( is\_member( $B[ $len - 1 ], @VN ) ) {

# If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)

$countChanged +=

myPush3( $followSets{ $B[ $len - 1 ] },

$followSets{ $B[0] } );

if ( ( $len - 2 > 0 )

and is\_member( 'ε', @{ $firstSets{ $B[ $len - 1 ] } } ) )

{

# If there is a production A → aBb, where FIRST(b) contains ε,

# then everything in FOLLOW(A) is in FOLLOW(B)

$countChanged += myPush3( $followSets{ $B[ $len - 2 ] },

$followSets{ $B[0] } );

}

}

}

}

return $countChanged;

}