Programming Language Descriptions and Implementations

BNF and context-free grammar

- Backus-Naur Form (1959)
  - way of specifying programming languages using formal grammars and production rules with a particular form of notation.
  - Invented by John Backus to describe Algol 58
  - BNF is nearly equivalent to context-free grammars

A Grammar for a small language

| : alternative
[ ]: optional
{ }: zero or more times

<program> → begin <stmt_list> end

<stmt_list> → <stmt> | <stmt>; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <integer> | <var> + <integer> | <var> - <integer>

<integer> → [-] <unsigned_integer>

<unsigned_integer> → <any_digit> | <non_zero_digit>{<any_digit>}

<any_digit> → 0 | <non_zero_digit>

<non_zero_digit> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

? What does this grammar describe?

Derivation Example

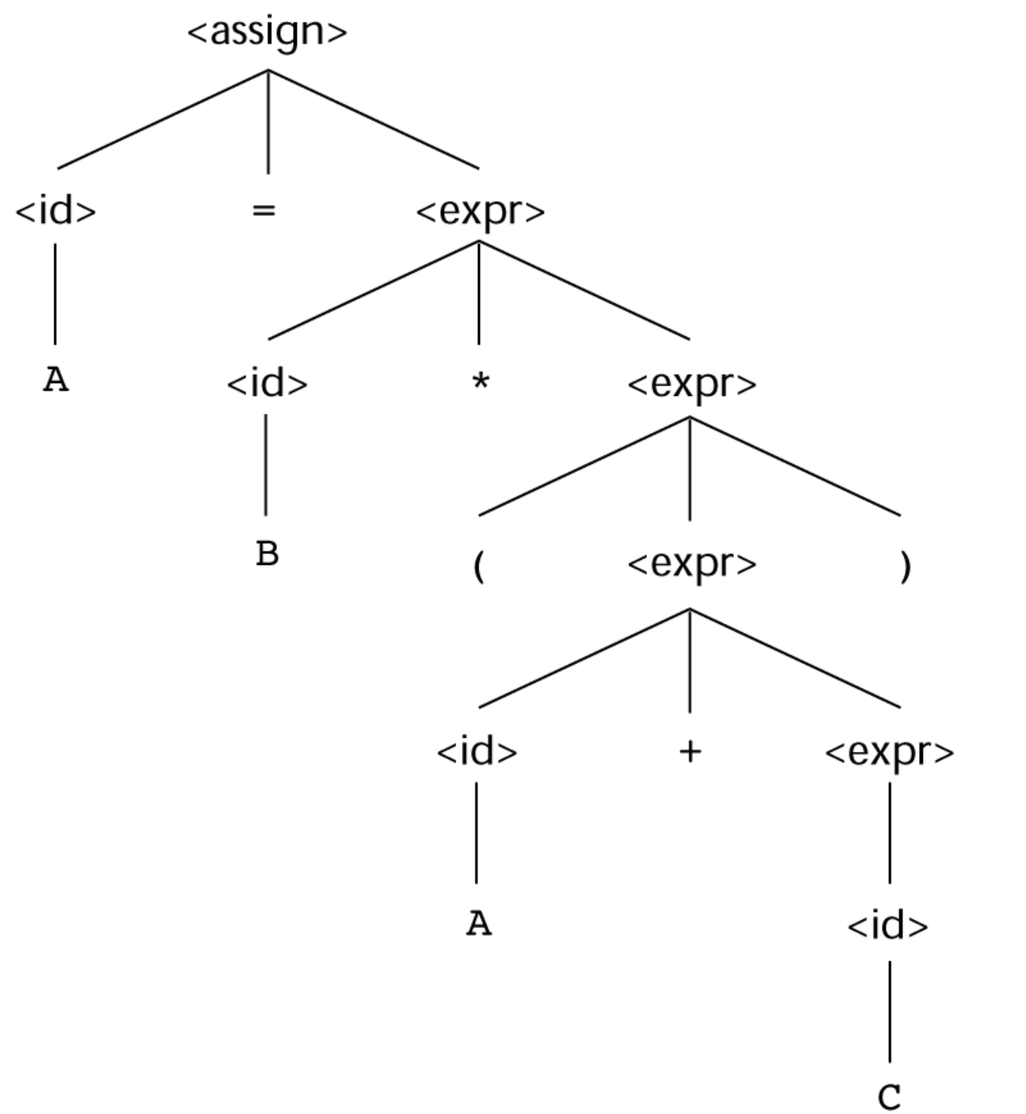
- <Program> => begin <stmt_list> end
  => begin <stmt>; <stmt_list> end
  => begin <var> = <expression>; <stmt_list>end
  => begin A = <expression>; <stmt_list> end
  =>begin A = <integer>; <stmt_list> end
  => begin A = <unsigned_integer>; <stmt_list> end
  => begin A = <non_zero_digit><any_digit>;<stmt_list>end
  => begin A = 8 <any_digit>; <stmt_list>end
  => begin A = 80;<stmt_list>end
  => begin A = 80; <stmt> end
  => begin A = 80; <var> = <expression> end
  => begin A = 80; B = <expression> end
  => begin A = 80; B = <integer> end
  => begin A = 80; B = <unsigned_integer> end
  => begin A = 80; B = <non_zero_digit> end
  =>begin A = 80; B = 2 end

Parse Tree

<assign> → <id> = <expr>
<id> → A | B | C | D
<expr> → <id> + <expr> | <id> * <expr>
| (<expr>) | <id>

A = B * (A + C)

<assign>
⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <id> * <expr>
⇒ A = B * <expr>
⇒ A = B * (<expr>)
⇒ A = B * (<id> + <expr>)
⇒ A = B * (A + <expr>)
⇒ A = B * (A + <id>)
⇒ A = B * (A + C)



Introduction

Two classes: static and dynamic semantics

- Static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms
  - Typed variables can only accept values of that type
  - A variable must be declared before it can be used
- Dynamic semantics express the meaning of the expressions, statements, and program.
  - After statement int x = 44 – y; x == 42 is trues

Attribute Grammars (Example)

- Syntax rule: <assign> → <var> = <expr>
  Semantic rule: <expr>.expected_type ← <var>.actual_type
- Syntax rule: <expr> → <var>[2] + <var>[3]
  Semantic rule:
  <expr>.actual_type ← if (<var>[2].actual_type = int) and
  <var>[3].actual_type = int)
  then int
  else real
  end if
  Predicate: <expr>.actual_type = <expr>.expected_type
- Syntax rule: <expr> → <var>
  Semantic rule:
  <expr>.actual_type ← <var>.actual_type
  Predicate: <expr>.actual_type = <expr>.expected_type
- Syntax rule: <var> → A | B | C
  Semantic rule:
  <var>.actual_type ← lookup(<var>.string)

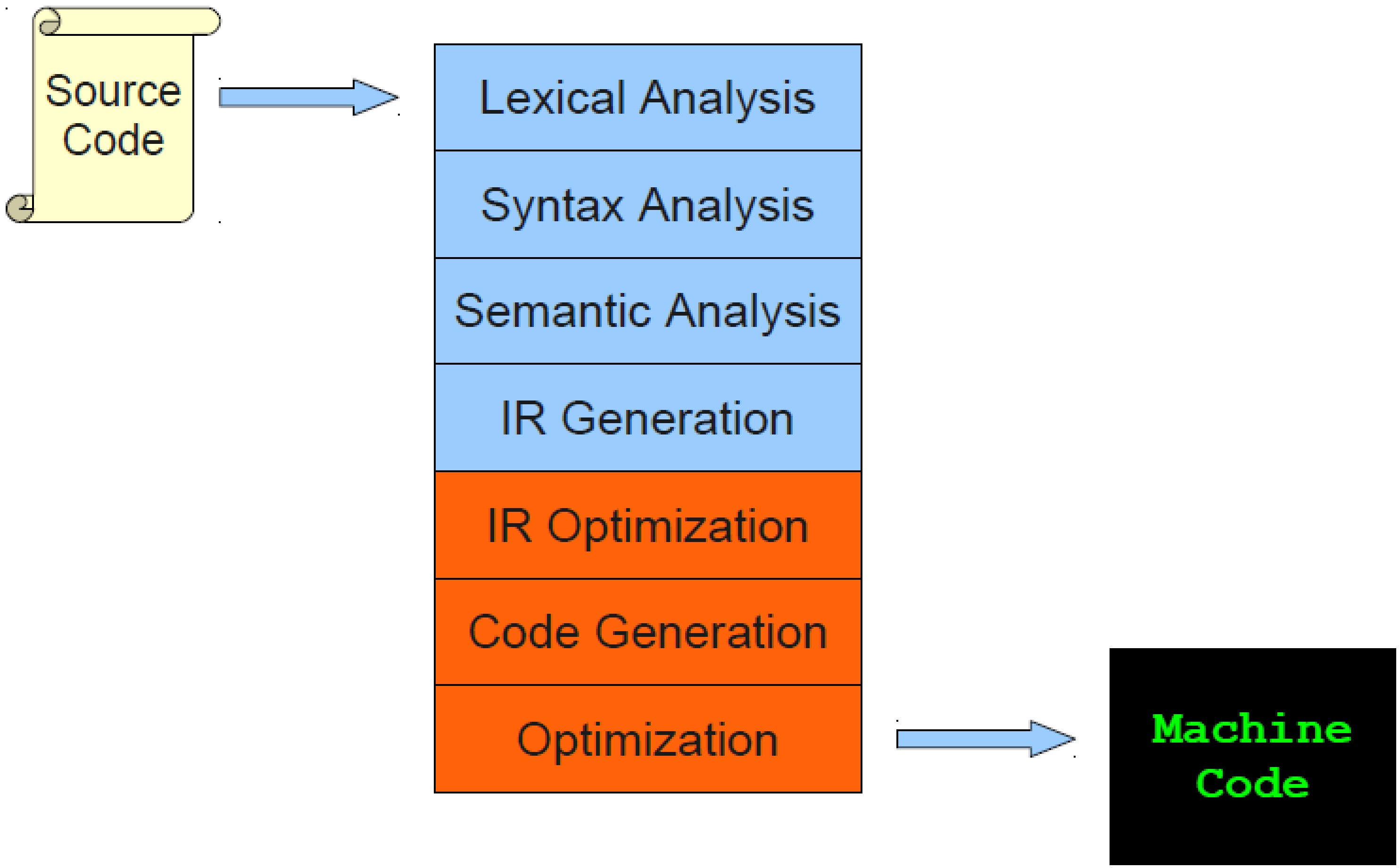
Blue: Basic BNF
Red: Semantic Func.
Green: Predicate Func.

Describing (Dynamic) Semantics

- There is no single widely acceptable notation or formalism for describing dynamic semantics

- Three formal methods:
  - ❖ Operational Semantics
  - ❖ Axiomatic Semantics
  - ❖ Denotational Semantics

The Structure of a Modern Compiler



IR: Intermediate Representation

Introduction

For all practical purposes, we can think of the source code as a very long string of characters from some alphabet.

Lexical analyzers collects characters into logical groupings (lexemes) and assigns internal codes (tokens) to the grouping.

As an analogy, we can think of the lexical analyzer as a machine that chops up source code into the individual “words”.

What lexical analyzers do

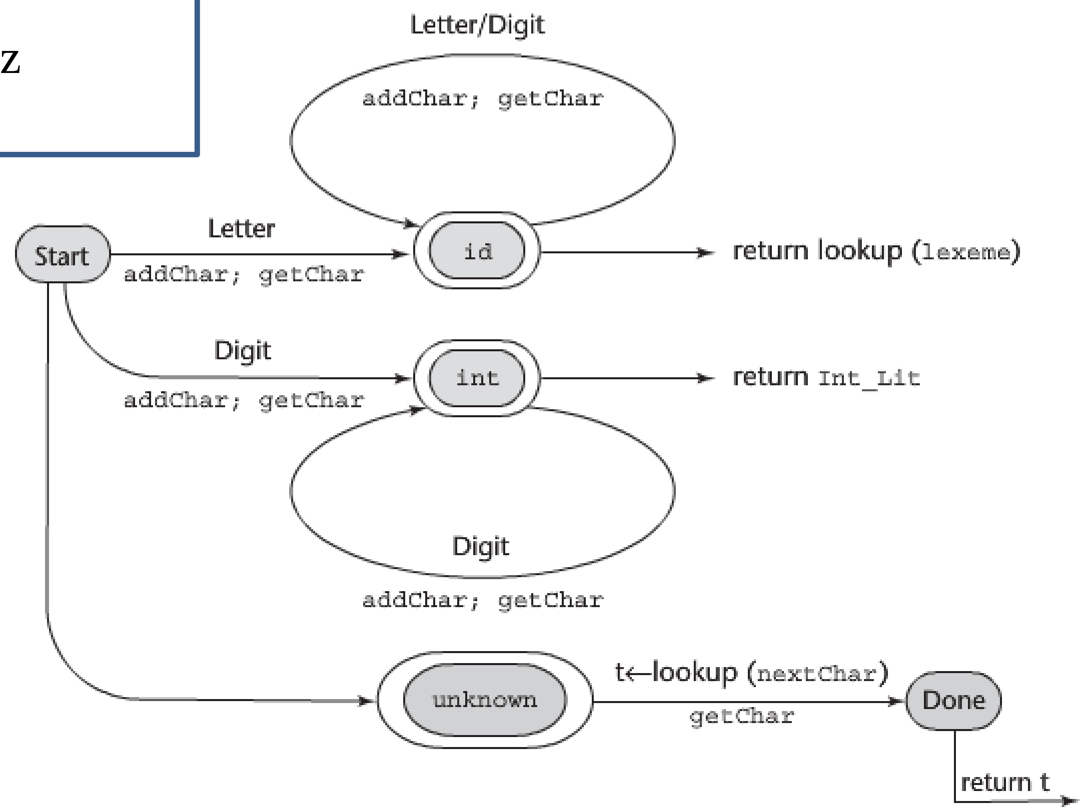
- The “front-end” for the parser
- A pattern matcher for character strings
- Identifies substrings of the source program that belong together => lexemes
  - Example: sum = B –5;

Lexeme	**Token**
sum	ID (identifier)
=	ASSIGN_OP
B	ID
-	SUBTRACT_OP
5	INT_LIT (integer literal)
;	SEMICOLON

State Transition Diagram: Example

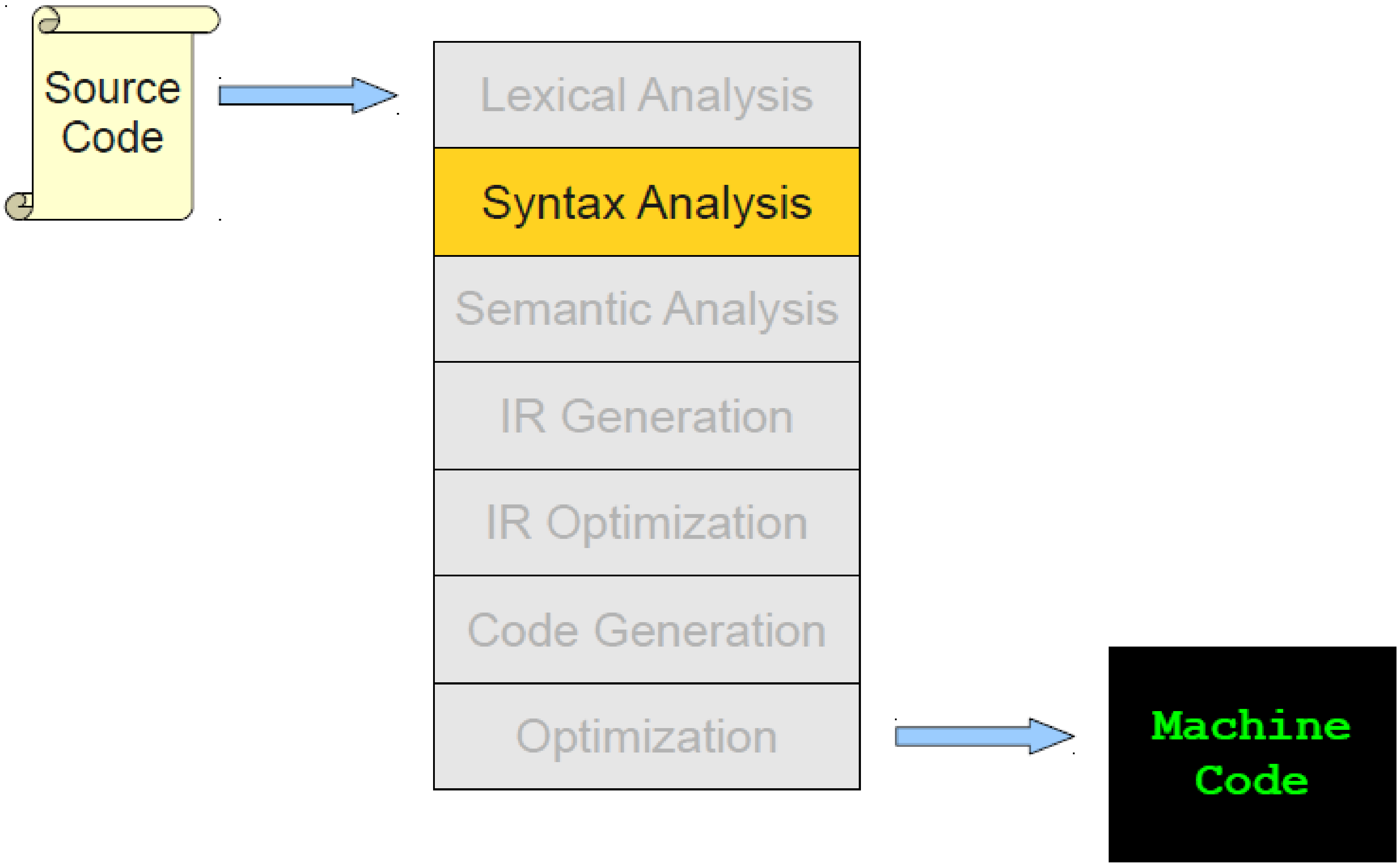
<id> → <letter>{<letter>|<int>} CFG
<int> → <digit>{<digit>}
<letter> → A | B | C |…| Z | a | b | …| z
<digit> → 0 | 1 | 2 | … | 9

Id -> letter(letter|digit)*
int->digit digit*
digit->[0-9]
letter->[a-zA-Z]
Regular grammar



Suppose we need a lexer that recognizes only names, reserved words, integer literals, parentheses. and arithmetic operators.

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

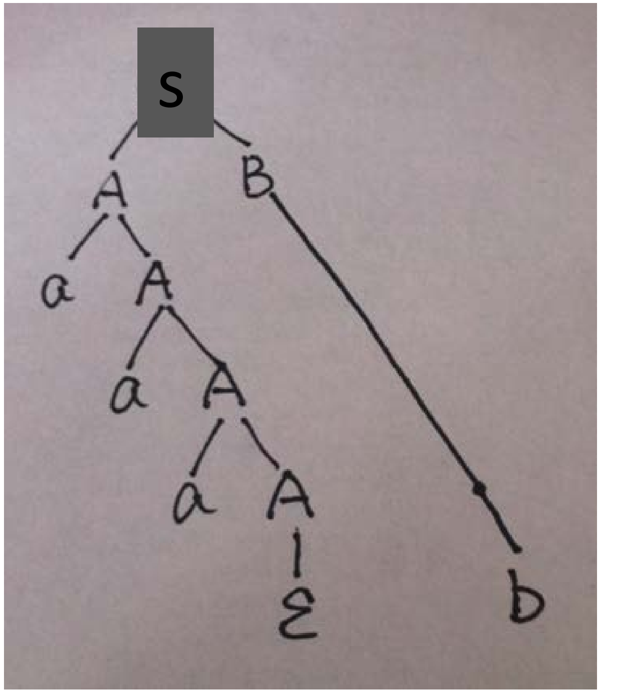
Introduction

- The syntax analyzer receives the source code in the form of tokens from the lexical analyzer and performs syntax analysis, which determines **the structure** of the program.

- Typically, the result generated by the syntax analyzer is a parse tree.

Top-down parsers

- Start with the start symbol and apply the productions until you arrive at the desired string.
  - Builds a parse tree in preorder
  - Corresponds leftmost derivation
  - LL parser



S	->	AB
A	->	aA | ε
B	->	b | bB

With this grammar

Here is a top-down parse of aaab

S	
AB	S -> AB
aAB	A -> aA
aaAB	A -> aA
aaaAB	A -> aA
aaaεB	A -> ε
aaab	B -> b

(Predicative) Top-down Parsers

From deviation aspect

- A nonterminal has more than one RHS
- The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

A -> bB | cBb | a

Example

<expr> → <term> {(‘+’ | ’-’) <term>}

<term> → <factor> {(‘*’ | ’/’)<factor>}

<factor> → id | ‘(‘<expr>‘)’

```
void expr()
{...}

void term()
{...}

void factor()
{...}

void error()
{...}
```

-- “id” here is not a terminal, but it stands for the identifier token

-- language literals are enclosed with ‘ ‘

```
Program 6:
read x
y = x + 5
write y
halt
```

```
else if (nextToken == READ) {
        lex();
        if (nextToken == IDENT) {
            printf("INP\nSTA %s\n", lexeme);
            lex();
        } else {
```

Homework 2

```
LMC assembly:
INP                     ; 1. Read a value from the input into the accumulator (A)
STA x                   ; 2. Store the value from the accumulator into memory location 'x'
LDA x                   ; 3. Load the value from memory location 'x' back into the accumulator
STA TMP0                ; 4. Store the accumulator value into a temporary memory location 'TMP0'
LDA NUM0                ; 5. Load the constant value '5' (NUM0) into the accumulator
ADD TMP0                ; 6. Add the value stored in 'TMP0' (original x) to the accumulator
STA y                   ; 7. Store the result (x + 5) into memory location 'y'
LDA y                   ; 8. Load the value of 'y' into the accumulator
OUT                     ; 9. Output the value of the accumulator (x + 5)
HLT                     ; 10. Halt the program

x    DAT 0              ; Memory location for variable x, initialized to 0
TMP0 DAT 0              ; Temporary memory to hold x, initialized to 0
NUM0 DAT 5              ; Memory location holding the constant 5
y    DAT 0              ; Memory location for variable y, initialized to 0
```

https://www.peterhigginson.co.uk/LMC/

Homework 2

Finding more than one error at time:

```
} else {
        error("Invalid assignment statement.");
        while (nextToken != IDENT && nextToken != READ &&
               nextToken != WRITE && nextToken != HALT && nextToken != EOF) {
            lex();
        }
        return;
}
```