Design of Compilers

LL1 Parser User Guide

By:

Dalia Ayman Ahmed

CESS pre-junior

Presented to:

Dr. Sahar Haggag

Eng. Amr Saad

program → stmt-seq

stmt-seq → stmt stmt-seq`
stmt-seq` → ; stmt-seq`
stmt-seq` → ε

stmt → if-stmt
stmt → repeat-stmt
stmt → assign-stmt
stmt → read-stmt
stmt → write-stmt

if-stmt → if exp then stmt-seq if-stmt`
if-stmt` → end
if-stmt` → else stmt-seq end

repeat-stmt → repeat stmt-seq until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp

exp → simple-exp exp`
exp` → comparison-op simple-exp
exp` → ε

comparison-op → <
comparison-op → =

simple-exp → term simple-exp`
simple-exp` → addop term simple-exp`
simple-exp` → ε

addop → +
addop → -

term → factor term`
term` → mulop factor term`
term` → ε

mulop → *
mulop → /

factor → (exp)
factor → number
factor → identifier

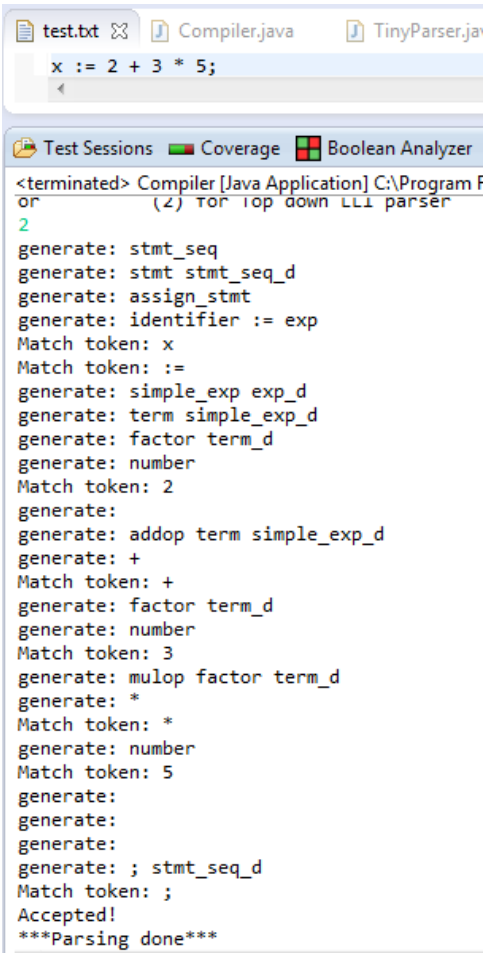
First Set of Non-Terminals:

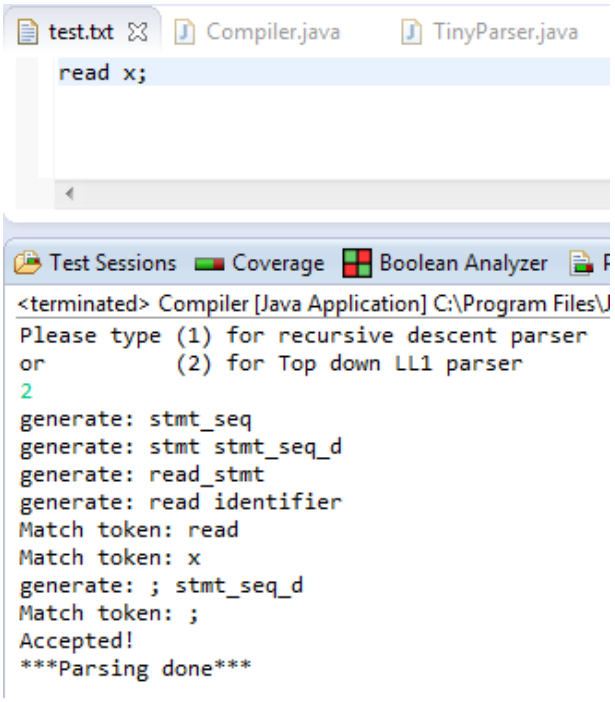
first(program) = {if, repeat, identifier, read, write}
first(stmt-seq) = {if, repeat, identifier, read, write}
first(stmt-seq`) = {;, ε}
first(stmt) = {if, repeat, identifier, read, write}
first(if-stmt) = {if}
first(if-stmt`) = {end, else}
first(repeat-stmt) = {repeat}
first(assign-stmt) = {identifier}
first(read-stmt) = {read}
first(write-stmt) = {write}
first(exp) = {(, number, identifier}
first(exp`) = {<, =, ε}
first(comparison-op) = {<, =}
first(simple-exp) = {(, number, identifier}
first(simple-exp`) = {+, -, ε}
first(addop) = {+, -}
first(term) = {(, number, identifier}
first(term`) = {*, /, ε}
first(mulop) = {*, /}
first(factor) = {(, number, identifier}

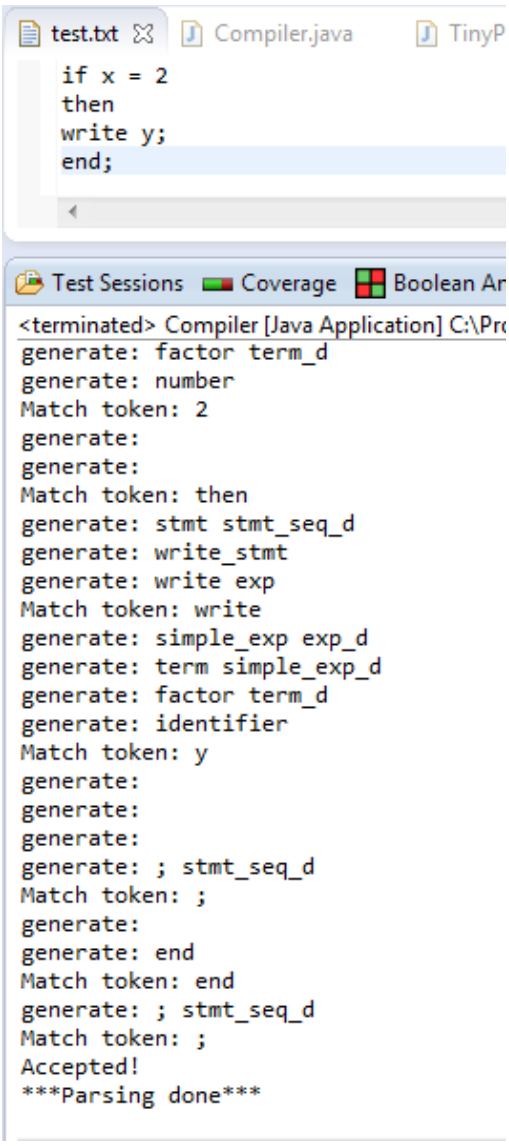
Follow Set of Non-Terminals:

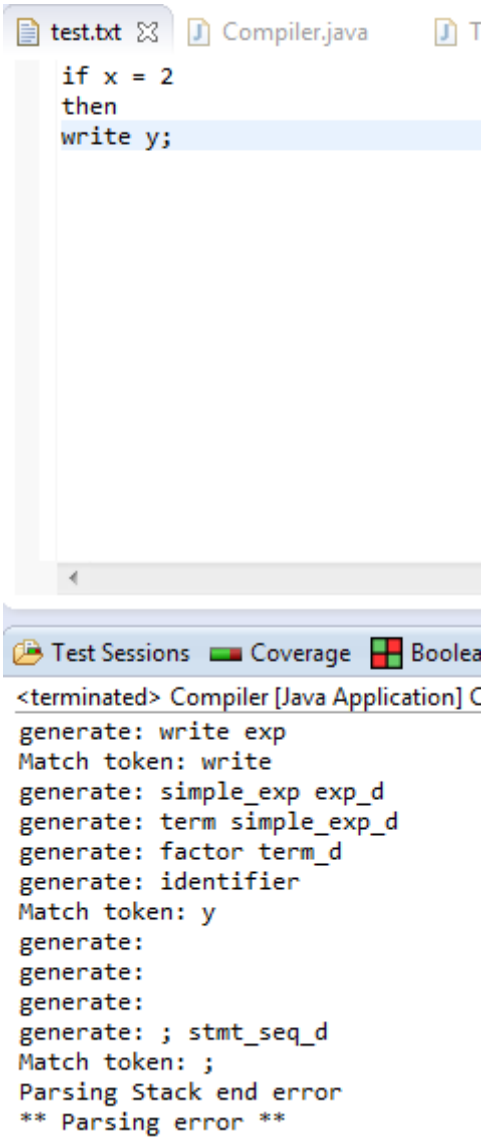
follow(program) = {$}
follow(stmt-seq) = {$, end, else, until}
follow(stmt-seq`) = {$, end, else, until}
follow(stmt) = {;, $, end, else, until}
follow(if-stmt) = {;, $, end, else, until}
follow(if-stmt`) = {;, $, end, else, until}
follow(repeat-stmt) = {;, $, end, else, until}
follow(assign-stmt) = {;, $, end, else, until}
follow(read-stmt) = {;, $, end, else, until}
follow(write-stmt) = {;, $, end, else, until}
follow(exp) = {then, ;, $, end, else, until,)}
follow(exp`) = {then, ;, $, end, else, until,)}
follow(comparison-op) = {(, number, identifier}
follow(simple-exp) = {<, =, then, ;, $, end, else, until,)}
follow(simple-exp`) = {<, =, then, ;, $, end, else, until,)}
follow(addop) = {(, number, identifier}
follow(term) = {+, -, <, =, then, ;, $, end, else, until,)}
follow(term`) = {+, -, <, =, then, ;, $, end, else, until,)}
follow(mulop) = {(, number, identifier}
follow(factor) = {*, /, +, -, <, =, then, ;, $, end, else, until,)}

Parsing Table: attached in an Excel file.
Screenshots:






test.txt
Compiler.java
TinyP
if x = 2
then
write y;
end;
Test Sessions
Coverage
Boolean Ar
<terminated> Compiler [Java Application] C:\Pr
generate: factor term_d
generate: number
Match token: 2
generate:
generate:
Match token: then
generate: stmt stmt_seq_d
generate: write_stmt
generate: write exp
Match token: write
generate: simple_exp exp_d
generate: term simple_exp_d
generate: factor term_d
generate: identifier
Match token: y
generate:
generate:
generate:
generate: ; stmt_seq_d
Match token: ;
generate:
generate: end
Match token: end
generate: ; stmt_seq_d
Match token: ;
Accepted!
Parsing done


test.txt
Compiler.java
T
if x = 2
then
write y;
Test Sessions
Coverage
Boolea
<terminated> Compiler [Java Application] C
generate: write exp
Match token: write
generate: simple_exp exp_d
generate: term simple_exp_d
generate: factor term_d
generate: identifier
Match token: y
generate:
generate:
generate:
generate: ; stmt_seq_d
Match token: ;
Parsing Stack end error
** Parsing error **