CS 4323
PROGRAMMING ASSIGNMENT #2
(PARSER)
Due: April 19, 2022

This project adds the second component of a compiler, the syntax analyzer (or parser), for Simple-Scala language to the scanner and other supporting programs constructed in the first project.

Refer to the syntax rules for simple-Scala language (A1), the *LL*(1) parse table for simple-Scala (A2), and the general structure of the *LL*(1) parser (A3). Add to your compiler a procedure/class PARSER() which, when given an arbitrary input program, generates its parse output, using the syntax rules for simple-Scala and the *LL*(1) parse table.

In order to write PARSER(), the first thing you have to do is to transform the syntax rules for simple-Scala (given in A1) into a PDA, which is nondeterministic in general. The transformation algorithm was given in class. Now, with the lookahead of one token allowed (note that a token is viewed as a single symbol here) and the *LL*(1) parse table (given in A2) added to the finite control, the PDA is a deterministic LL(1) parser.

PARSER() should be called *once* from the main body of your program. Once it is called from the main body, the whole parsing must be done in PARSER().

SCANNER() needs some modification now. First, it will not be called from the main body (see A3); it will be called from PARSER() when a lookahead token is needed. Second, your previous SCANNER() *printed* a token as it identified one; here, it must *pass* it to PARSER(). SCANNER() will pass it in the form of a pair (token, token type), where token means the actual token identified (i.e., lexeme) and token type means the classification of tokens into those represented in the syntax rules. Finally, SCANNER() need not print out the scanning information now, since you know that it does the correct job (if you correctly constructed SCANNER() in the first programming assignment).

It is convenient to use *integer codes* to classify token types. We shall use integers 1 and 2 for [*id*] and [*const*], integers 3..30 for the 28 reserved keywords, and integers 31..41 for the 11 special symbols.

It is also convenient to define the stack of the parser using an array of integers; an array of size 100 should be enough for our purpose. Note that the stack stores nonterminals and terminals. For terminal tokens, use the integer codes 1..41 passed from SCANNER(). Use integers 42..69 for the 28 nonterminals in the grammar. Use 0 for the stack bottom marker.

More details will be discussed in class, with examples. For the required output, the following information must be printed in a well-documented form:

- Print out the source program to be parsed (given on the next page), exactly as you stored in your input file. This must be done before parsing begins.
- Print out the parse output, i.e., the sequence of actions taken by the parser together with supporting information (stack-top symbols and lookhead tokens in their original names and integer codes). An example of the parse output will be given in class; it is important to follow the output format given in this example.
- Print out the SYMTAB content. This must be done in the main body of your compiler after parsing has been completed.

Source program:

```
package a;
package b;
import a.xyz; import b.c…67;
abstract class {
val a, b, c : real;
def x (y, w) { y <= w; };
while (not ( true or false)) return (47 * (x + 25));
}
protected object {
val i, j, k : int;
if (@ x 25) case i = j + k * 5 => print (i);
else in (i, j, k);
}
private class {
val tt, ff: bool;
return (not (true or @ x 5) and false);
}
$
```

Submit your program and output on or before the due date. An incomplete work claiming partial credit (of no more than half the full credit) should include a description of what troubles you had. Failure to include this information may result in zero credit on the work. Penalty for late submission: 20% per calendar day.

Assignment handed out on March 24, 2022.

A1. Syntax Rules for Simple-Scala Language

[1] <scala> → <packages> <imports> <scala-body>
[2]-[3] <packages> → package [id] ; <packages> | ε
[4]-[5] <imports> → import [id] ; <imports> | ε
[6]-[7] <scala-body> → <subbody> <scala-body> | ε
[8] <subbody> → <modifier> <subbody-tail>
[9]-[13] <modifier> → abstract | final | sealed | private | protected
[14] <subbody-tail> → <tail-type> <block>
[15]-[16] <tail-type> → class | object
[17] <block> → { <stmts> }
[18]-[19] <stmts> → <stmt> ; <stmts> | ε
[20]-[28] <stmt> → <dcl> | <asmt> | <if> | <while> |<case>
| <in> | <out> | <return> | <block>
[29]-[30] <dcl> → val <dcl-tail> | def [id] (<ids>) <block>
[31] <dcl-tail> → <ids> : <type>
[32] <ids> → [id] <more-ids>
[33]-[34] <more-ids> → , [id] <more-ids> | ε
[35]-[37] <type> → int | real | bool
[38] <asmt> → [id] <= <expr>
[39] <if> → if (<expr>) <stmt> ; else <stmt>
[40] <while> → while (<expr>) <stmt>
[41] <case> → case [id] = <expr> => <stmt>
[42] <in> → in (<ids>)
[43] <out> → print (<ids>)
[44] <return> → return (<expr>)
[45]-[46] <expr> → <arith-expr> | <bool-expr>
[47]-[49] <arith-expr> → [id] <arith> | [const] <arith> | (<arith-expr>) <arith>
[50]-[52] <arith> → + <arith-expr> | * <arith-expr> | ε
[53]-[56] <bool-exp> → not (<bool-expr>) <bool> | true <bool>) | false <bool>
| @ <arith-expr> <arith-expr>
[57]-[59] <bool> → and <bool-expr> | or <bool-expr>