Building a Parser for SL (A2)

Note: This assignnment can be done in groups of two. You should build your parser(s) with the tool mentioned in the description of the assignment in either C/C++ or Java langauge. The deadline for the assignment submission is Thursday, December 14, 2023 on or before 9:00 a.m. via email (masindhu@qau.edu.pk) with names of the group members in the subject line of email along with assignment name. A viva will be conducted in the subsequent class starting at 0905 a.m.

1 Assignment Introduction

You already know that given a context-free grammar G and a string T, a parser can do the following: Decide whether G can generate T (i.e. "T is in the grammar G"). Produce a parse tree of T, which shows how to rewrite the start symbol S in G to produce T. Your job will be to construct a parser to do both of the above tasks. Your program will take as its input a program in SL; if the program is syntactically valid, your parser will produce a parse tree for it. We will grade your program by feeding it certain SL programs--some valid, some not--and check that your program outputs a valid parse for syntactically valid programs, and a meaningful error message for invalid programs.

2 Grammar for SL

In addition to the information you got in the last assignment, let us examine the context-free grammar (Σ, V, S, $\rightarrow$) for SL: Σ = { PROGRAM, FUNCTION, PARAMLIST, PARAMREST, BLOCK, STATEMENT, BREAK, CALL, ARGLIST, ARGREST, IF, ELSE, LET, READ, RETURN, WHILE, WRITE, EXPR, BINOP, UNOP, number, identifier, function, break, call, if, else, let, read, return, while, write, (,), {, }, „ ;, =, +, -, *, /, %, <, >, <=, >=, ==, !=, &, |, ~, !}

V = { number, identifier, function, break, call, if, else, let, read, return, while, write, (,), {, }, „ ;, =, +, -, *, /, %, <, >, <=, >=, ==, !=, &, |, ~, !}

S = PROGRAM

$\rightarrow$ is given by:

- PROGRAM $\rightarrow$ FUNCTION* BLOCK
- FUNCTION $\rightarrow$ function identifier (PARAMLIST) BLOCK
- PARAMLIST $\rightarrow$ identifier PARAMREST | e
- PARAMREST $\rightarrow$, identifier PARAMREST | e
- BLOCK $\rightarrow$ { STATEMENT* }
- STATEMENT $\rightarrow$ BREAK | CALL ; | IF | LET | READ | RETURN | WHILE | WRITE
- BREAK $\rightarrow$ break ;
- CALL $\rightarrow$ call identifier (ARGLIST)
- ARGLIST $\rightarrow$ EXPR ARGREST | e
- ARGREST $\rightarrow$, EXPR ARGREST | e
- IF $\rightarrow$ if EXPR BLOCK ELSE

- ELSE → else BLOCK | e
- LET → let identifier = EXPR ; | let identifier = CALL ;
- READ → read identifier ;
- RETURN → return EXPR ;
- WHILE → while EXPR BLOCK
- WRITE → write EXPR ;
- EXPR → number | identifier | (EXPR) | (UNOP EXPR) | (BINOP EXPR EXPR)
- BINOP → + | - | * | / | % | & | | | < | > | <= | >= | == | !=
- UNOP → ˜ | !

3 Building the Parser

You will build the parser for the above grammar and rules using the Bison parser generator tool which can be retrieved from the link (`https://www.gnu.org/software/bison/manual/bison.html`). You can generate the parser code using Bison for several languages as described in the given link. You can choose for example C or Java for your assignment. Your parser must be able to ouput the parse tree of a valid SL program in a comprehensible manner to a person famailiar with tree traversal methods you studied in your data structure course. The mentioned link provides examples on how to write grammars/rules to generate parsers using Bison you can go through them when needed. Your parser should also be equipped with error recovery mechanism to report an error when there is one for this you will need to go through Section 3.5 on the given link to report problems line numbeer(s) for example and other useful information to the user about the nature of the error. You should also be able to answer the question which parsing algorithm is being used by Bison at the back-end.