

Grammar

Write a recursive descent parser by hand for the following grammar:

$S \rightarrow \text{WHILE-S} \mid \text{ASSIGN-S}$
 $\text{WHILE-S} \rightarrow \text{while CONDITION S}$
 $\text{ASSIGN-S} \rightarrow \text{identifier} = \text{constant}$
 $\text{CONDITION} \rightarrow \text{identifier} (< \mid > \mid =) \text{constant}$

Assumptions

- Token structure:
 - token.kind has type of token
 - Type of token is NAME_T
- Scanner functions:
 - Token nextToken() returns next token.
 - returnToken() puts token back onto stream.
- ParseTree structure:
 - ParseTree newTree(type, other params) returns a parse tree with all the parameters inside it.
 - Types can be WHILE_P, ASSIGN_P, CONDITION_P
 - You can cast Tokens into ParseTrees.
- **Program does not contain errors.**

Approach:

Define one Boolean function for each non-terminal with one ParseTree tree parameter passed by reference.

- If parsing succeeds, tree contains new parse tree, and function returns true
- If parsing fails, tree doesn't change and function returns false.

Parser:

```
Boolean S (*ParseTree tree)
{
    return (WHILE_S(tree) || ASSIGN_S(tree));
}

Boolean WHILE_S(*ParseTree tree)
{
    Token while_t;
    ParseTree condition, body;
    While_t = nextToken();
    if (while_t.kind != WHILE_T)
    {
        returnToken();
        return false;
    }
    CONDITION (&condition);
    S(&body);
    tree = newTree(WHILE_P, while_t, condition, body);
    return true;
}

Boolean ASSIGN_S(*ParseTree tree)
{
    Token lhs, eq, rhs;
    lhs = nextToken();
    if (lhs.kind != IDENTIFIER_TOK)
    {
        returnToken();
        return false;
    }
    eq = nextToken();
    rhs = nextToken();
    tree = newTree(ASSIGN_P, lhs, eq, rhs);
    return(true);
}

Boolean CONDITION (*ParseTree tree)
{
    tree = newTree( CONDITION_P,
        nextToken(), nextToken(), nextToken());
    return true;
}
```