Andrew Salopek
3361
11/19/2015

Project 2 Report

Intro/Data Structures

This is a recursive descent parsing program. It reads a program and annotates it in XML-format. In my program, I will be using a counter, indentCtr, to keep track of, and to implement the XML formatting. The counter will increase with every nonterminal function call, and decrease at the end of the function. The pseudo-code is derived from the class slides (2.3)

Pseudo-Code

Parser

```
Input
    1. A program
    2. The context free grammar
Output
    1. XML parse of input program if the program follows the grammar;
       otherwise report parsing error
```

Main function

```
Data
    input_token             //current token of the input program. Global
Plan
        //get the first token
        input_token = scan()
        return (program())
    }
```

Program()

```
Input
    1. inputToken
Output
    1. Returns ok if the input of the program follows the production on
       <program>; and returns parse_error otherwise
Plan
    program(){
        //<program> -> <stmt_list> $$
        //$$ is the end of the program token
        case input_token of
            id, read, write, $$:     //these tokens are chosen because
                                     //they are able to be matched to
```

```
                            //non-terminals
                    if (stmt_list()==ok)
                        //rest of program must be "end of program"
                        indent
                        match($$)
                        return ok
                    else return parse_error
                otherwise: return parse_error
            }
```

Match(expected)

Input

1. A token expectedToken, inputToken

Output

2. Ok and gets the next token into input_token if the expected is the same as input_token; otherwise output parse_error

Plan

```
            {
                if (expectedToken == input_token)
                    //get next token from the input program
                    if token == read
                        print XML
                    input_token = scan()
                    return ok
                else return parse_error
            }
```

Stmt_list()

//<stmt_list> -> <stmt> <stmt_list> | ε

Input

1. No input

Output

1. Returns ok if the input program follows the production on <stmt_list>; and returns parse_error otherwise.

Plan

```
            {
                case input_token of id, read, write:
                    if (stmt() == ok)
                        if (stmtList==ok)
                        indent
                        return ok
                    else return parse_error
```

```
        }
```

Stmt()

//<stmt> -> id := <expr> | read id | write <expr>

Input

1. None

Output

1. Returns ok if the input program follows the production on <stmt>; and returns parse_error otherwise.

Plan

```
        stmt(){
                case input_token of
                        id:
                                indent
                                match(id)
                                if match (:=)
                                if(expr()==ok)
                                        indent
                                        return ok
                                else return parse_error
                        read:
                                indent
                                match(read)
                                match(id)
                                        indent
                                        return ok
                                else return parse_error
                        write:
                                indent
                                match(write)
                                if(expr()==ok)
                                        indent
                                        return ok
                                else return parse_error
                        otherwise return parse_error
        }
```

expr()

Input

1. None

Output

1. returns ok if the input program follows the production on <expr>; and returns parse_error otherwise.

Plan
```
{
    case input_token of lparen, id, number:
        if(term_tail == ok)
            indent
            if(term()==ok)
                indent
                return ok
        else return parse_error
    otherwise return parse_error
}
```

Term_tail()

Input
1. none

Output
1. returns ok if the input program follows the production on <term_tail>; and returns parse_error otherwise.

Plan
```
{
    case input_token of +,-:
        indent
        if(addOp()==ok)
            if(term()==ok)
                if(termTail==ok)
                    indent
                    return ok
        else return parse_error
    case input_token of rparen, id, read, write, $$:
        indent
        return ok
        else return parse_error
    otherwise return parse_error
}
```

Term()

Input
1. none

Output
1. returns ok if the input program follows the production on <term>; and returns parse_error otherwise.

Plan
```
{
        case input_token of lparen, id, number:
            if(factor()==ok)
                if(factorTail()==ok)
                    indent
                    return ok
            else return parse_error
        otherwise return parse_error
    }
```

Factor_tail()

Input

1. none

output

1. returns ok if the input program follows the production on <factor_tail>; and returns parse_error otherwise.

Plan

```
    {
        case input_token of *,/:
            if(multOp()==ok)
                if(factor()==ok)
                    (factorTail() == ok)
                        indent
                        return ok
            else return parse_error
        case input_token of +,-,),id,read,write,$$:
            indent
            return ok
        otherwise return parse_error
    }
```

Factor()

Input

1. none

Output

1. returns ok if the input program follows the production on <factor>; and returns parse_error otherwise.

Plan

```
    {
        case input_token of (:
            match(lparen)
            if(expr()==ok)
                indent
                match(rparen)
```

```
                        indent
                        return ok

                case id:
                        indent
                        match(id)
                        return ok
                case number:
                        indent
                        match(number)
                        return ok
                otherwise return parse error
        }
```

Add_op()

Input
1. none

output
1. returns ok if the input program follows the production on ,<add_op>; and returns parse_error otherwise.

Plan

```
        {
                case input_token of +
                        indent
                        match(+)
                        return ok
                case input_token of –
                        indent
                        match(-)
                        return ok
                otherwise return parse_error
        }
```

mult_op()

Input
2. none

output
2. returns ok if the input program follows the production on ,<mult_op>; and returns parse_error otherwise.

Plan

```
        mult_op(){
                case input_token of *
                        indent
                        match(*)
                        return ok
                case input_token of /
                        indent
                        match(/)
                        return ok
                otherwise return parse_error
        }
```

Test Cases

For test cases, I will use a combination of different tokens. I will also make expressions incomplete in order to demonstrate the error in parsing. I have included print statements that print the return status of SOME nonterminal functions.

Acknowledgments

I did not work with anybody on this project, I have only consulted the class slides for a basis of my pseudo-code. I have learned how to recursively traverse an input program, and trace the parse in an XML format