Indian Institute Of Technology Jammu
Department of Computer Science

Programming Languages

Assignment 6

Vineet Bakshi
2018UCS0066

Supervisors:
S. Arun Kumar

IIT Jammu, March 2020

About

This is the design document for various components of a compiler starting with a simple version of while language as the source language and converting it into 3AC and then executing it. The various stages are Lexical Analysis, Parsing, Semantic Analysis, 3 Address Code generation, execution. The algorithms I've used are designed to be implemented in SML, hence are recursive. I've numbered some lines of code in some chapters to show that they are related to the respective chapter.

Table of Contents

1. Lexical Analysis

1.1 Token Structure

ID stands for identifier. OP is operator. The other two fields in tokesn are to provide line number and characters read up to that token. PUN stands for punctuation. The record of a particular identifier corresponding to a identifier in the symbol table can be fetched just using the lexeme of the identifier since it can uniquely identify a record because the while language only has one main scope and no nested scopes. I have stamped the tokens with line numbers and characters other than the lexeme so that I don't have to maintain them in the symbol table.

```
datatype token = RES of string*int*int
    | ID of string*int*int
    | INT of string*int*int
    | BOOL of string*int*int
    | PUN of string*int*int
    | BoolOP of string*int*int
    | AddOP of string*int*int
    | MulOP of string*int*int
```

1.2 Algorithm for scanning

SML is a functional language, it has no memory so the following items which need to be updated throughout the process of scanning need to be passed recursively as arguments. Since this is tedious I have used shorthand for the arguments wherever possible. Also "Call Lexer" is a shorthand for "call and pass these parameters".

Parameters of function Lexer:

- Source File (Containing the list of characters in source file that are not read)
- Symbol Table
- Token List (Consists of tokens made so far)
- Line Number
- Characters read until now First element of the source file is peek and second is nextEl.

Algorithm

```
Lexer (...arguments...){
    peek = sourceFile[0]
    nextEl = sourceFile[1]
    switch( peek ) {
    case '&':
    if(nextEl='&'){
    //Update Arguments
        sourceFile=sourceFile[2 to last];
        tokens = tokens@[BoolOP("&&",line,chars)]
        chars=chars+2;
    }else
    raise Syntax Error;
    return(call Lexer)
    case '|':
```

```
if( nextEl='|' ){
//Update Arguments
    sourceFile=sourceFile[2 to last];
    tokens = tokens@[BoolOP("||",line,chars)]
    chars=chars+2;
}else raise Syntax Error
return(call Lexer)
case '=':
if( nextEl = '=' ){
//Update Arguments
    sourceFile=sourceFile[2 to last];
    tokens = tokens@[BoolOP("==",line,chars)]
    chars=chars+2;
}else{
//Update Arguments
    sourceFile=sourceFile[1 to last];
    tokens = tokens@[BoolOP("=",line,chars)]
    chars=chars+1;
}return(call Lexer)
case '!':
if( nextEl = '=' ){
//Update Arguments
    sourceFile=sourceFile[2 to last];
    tokens = tokens@[BoolOP("!=",line,chars)]
    chars=chars+2;
}else{
//Update Arguments
    sourceFile=sourceFile[1 to last];
    tokens = tokens@[BoolOP("!",line,chars)]
    chars=chars+1;
}return(call Lexer)
case '<':
if( nextEl = '=' ){
//Update Arguments
    sourceFile=sourceFile[2 to last];
    tokens = tokens@[BoolOP("<=",line,chars)]
    chars=chars+2;
}else{
//Update Arguments
    sourceFile=sourceFile[1 to last];
    tokens = tokens@[BoolOP("<",line,chars)]
    chars=chars+1;
}return(call Lexer)
case '>':
if( nextEl = '=' ){
//Update Arguments
    sourceFile=sourceFile[2 to last];
    tokens = tokens@[BoolOP(">=",line,chars)]
    chars=chars+2;
}else{
//Update Arguments
    sourceFile=sourceFile[1 to last];
    tokens = tokens@[BoolOP(">",line,chars)];
    chars=chars+1;
}return(call Lexer)
}//switch ends here
/*The above block checks for boolean operators and tokenises them according to
Token structure.*/
if (digit(peek)){//Digit is a utility function that returns true if peek is a
digit
    val digitVal,charUsed,newSourceFile = (getDigit sourceFile);/*The function
    getDigit is a Utility Function which concatenates all digits ahead of peek
    and returns it with the new sourceFile and no. of characters used*/
//UpdateArguments
    sourceFile=newSourceFile
    tokens = tokens@[INT(digitVal,line,chars)]
    chars=chars+charsUsed;
return(call Lexer)
}
if (peek is a character){//Done using Utility function isChar
    val id,charsUsed,newSourceFile = (getString sourceFile)/*Just like getDigit,
    returns the concatenated string and the appropriate arguments*/
    //UpdateArguments
    sourceFile=newSourceFile
    if(isReserved(id))//Utility Function that checks if the string is a reserved
    Keyword.
        tokens = tokens@[RES(id,line,chars)];
    else{
        if (id="true" or "false"){
        tokens=tokens@[BOOL(id,line,chars)]
        chars=chars+charsUsed;
        return(call Lexer)
        }
        tokens = tokens@[ID(id,line,chars)]
        chars=chars+charsUsed;
    return(call Lexer);
    }
}
if(peek is arithmetic operator){
//UpdateArguments
    sourceFile=SourceFile[1 to last]
    tokens = tokens@[ArithOP(peek,line,chars)]
    chars=chars+1;
    return(call Lexer)
}
if(peek is newline,tab,or space){
    switch(peek){
        case(newline):
            line+=1
            return(call Lexer)
        case(space or tab):
            chars+=1
            return(callLexer)
    }
}
if(sourceFile=nil)//Complete program is read
return(..arguments...);
//Anything else are punctuation Symbols
```

```
    //UpdateArguments
        sourceFile=sourceFile[1 to last]
        tokens = tokens@[PUN(peek,line,chars)]
        chars=chars+1;
    return(call Lexer)
}
```

2. Error Handling

I have maintained a seperate file containing a structure to raise errors wherever necessary. Errors are raised by calling the errorFn which takes tokens and exceptions as arguments, extracts line number and characters from them and then prints them with the desired error.

```
signature errors =
sig
        exception SyntaxError
        exception PreviouslyDeclaredVariable
        exception UndeclaredVaiable
        exception UninitializedVariable
        exception TypeMismatch
        val errorFn = fn : token -> string -> int
end
```

3. Parsing

3.1 Grammar Used

I have used the following grammar for declarations, assignments, loops, and if else constructs (written as EBNF). This grammar is left factored and is fit for a **recursive descent** parser using a single lookahead.

Program ::= “program” *Identifier* “::” *Block* .
Block ::= *DeclarationSeq* *ComandSeq* .
DeclarationSeq ::= *Declaration* .
Declaration ::= “var” *VariableList* “:” *Type* “;” .
Type ::= “int” | “bool” .
VariableList ::= *Variable* [“,” *VariableList*] .
ComandSeq ::= ”{” *Command* ”;” ”}” .
Command ::=
Variable “:=” *Expresssion* |
“read” *Variable* |
“write” *IntExpression* |
“if” *BoolExpression*
“then” *ComandSeq*
“else” *ComandSeq*
“endif” |
“while” *BoolExpression*
“do”
ComandSeq
“endwh”

The EBNF given in slides for the expression is left recursive and can’t be used for a recursive descent parser. So I left factored the production rules and made them such that the first sets of any two different non-terminals are different.

Expression ::= IntExpression | BoolExpression
For **integer** expressions
IntExpression ::= IntTerm [anotherIntExpression] .
anotherIntExpression ::= AddOp IntTerm [anotherIntExpression] .
IntTerm ::= IntFactor [anotherIntTerm] .
anotherIntTerm ::= MulOp IntFactor [anotherIntTerm] .
IntFactor ::= Numeral | Variable | “(” IntExpression “)” | “~” IntFactor .
For **boolean** expressions
BoolExpression ::= BoolTerm [anotherBoolExpression] .
anotherBoolExpression ::= “| |” BoolTerm [anotherBoolExpression] .
BoolTerm ::= BoolFactor [anotherBoolTerm] .
anotherBoolTerm ::= “&&” BoolFactor [anotherBoolTerm] .
BoolFactor ::= “true” | “false” | Variable | “(” BoolExpression “)” | “~” BoolFactor | Comparison.
Comparison ::= IntExpression RelOp IntExpression .
RelOp ::= “<” | “<=” | “=” | “>” | “>=” | “< >” .
Variable ::= Identifier .

Note:Non-terminals AddOp, MulOp, RelOp, Identifier and Numeral accept tokens of addition operators, multiplication operators, boolean operators , ID(”string”) and INT(”string”) respectively.

3.2 Parsing Algorithm

After writing appropriate grammar the algorithm is merely a translation of the production rules to function. First the expression calls the IntExpression or the BoolExpression based on the type of lvalue. After that each token executes a particular production rule based on the starting token of the production rule.The parsing algorithm will therefore constitute of a function for every non-terminal which runs its production rule. By run I mean it calls the corresponding function of every non-terminal in it's rule and calls the match function for any terminal. The match function checks if the lookahead is the same as a particular token and loads the next token into lookahead. **Note: Since writing algorithms for individual non-terminals is tedious and repetitive I will just write the algorithm for important ones. Also, Since the functions in sml are recursive I had to pass the arguments viz. Tokenlist and SymbolTable to the functions and also return them.**

```
//Since my match function is checking just the lexeme it doesn't matter what
I send in the latter two fields of the token.
match (lookahead::tokenList) ExpectedToken{
    if (lexeme of lookahead=lexeme of ExpectedToken)
        return tokenList
    else
        errorFn(lookahead,"SyntaxError");
}

/*Each function for non-Terminal must return the tokenList after its
processing*/

IntExpression (tokenList){
return (anotherIntExpression (IntTerm tokenList)) /*First calls IntTerm
and then anotherIntExpression in accordance with its production rule*/
}

anotherIntExpression (lookahead::tokenList){
    switch(lookahead){
        case (AddOP('+',0,0)): newlist = match tokenList AddOP('+',0,0)
        case (AddOP('-',line,chars)): newlist = match tokenList AddOP('-',0,0)
        default : return(lookahead::tokenList) /*Do nothing corresponding
        to epsilon production*/
    }
    newlist = (anotherIntExpression (IntTerm newlist))
    return newlist;
}
```

3.3 Parsing boolean expressions

The following production rule
BoolFactor ::= “true” | “false” | Variable | “(” BoolExpression “)” | “˜” BoolFactor | Comparison.
Can't directly be parsed by a recursive descent paser because given a token say ID(”x”) it can't differentiate between production rule Variable and Comparison as both might start with ”x”.Similar is the conflict between ”(” BoolExpression ”)” and Comparison.

3.3.1 Variable vs Comparison

The semantic analyser comes to our rescue. Since the variable declaration stage comes before any expression evaluation, we can infer which production rule to use from the type of identifier. If it is a bool then run Variable else run Comparison. Also run Comparison if you get an INT(num) token.

3.3.2 “(” BoolExpression “)” vs Comparison

This case is different as we might encounter n number of left paranthesis before encountering first numeral or identifier.So I made a utility function that returns first token encountered after n brackets and used the type of this token to choose between the productions.

```
BoolFactor(lookahead::tokens,symTable){
    switch(lookahead){
        case(INT(num,line,chars)) : return((Comparison lookahead::tokens symTable))
        case(ID(id,line,chars)) :
        if((checkIfPresent id symTable)=false)
            errorFn(lookeahead,"UndeclaredVariable");
        else{
            if(getType(lookahead)="int")
                return((Comparison lookahead::tokens symTable));
            else
                return((Variable lookahead::tokens symTable));
        }
        case(PUN("(",line,chars)):
            switch(firstTokenAfterN tokens):
                case(INT(num,line,chars)) : return((Comparison lookahead::tokens symTable))
                case(ID(id)):
                    if(getType(lookahead)="int")
                        return((Comparison lookahead::tokens symTable));
                    else
                        return((BoolExpression tokens symTable));
        case(BoolOP("!",line,chars)):
            return (BoolFactor tokens symTable);
        case(Bool(value)):
            return (tokens,symTable)
        default:
            errorFn(lookahead,"SyntaxError");
    }
}
```

4. Symbol Table

4.1 Signature

I have used list of records to implement symbol table. I am only storing the identifiers in the symbol table as I am passing the line number and characters itself with the tokens (refer token structure).Also since the name of identifiers is unique I have used them to refer to a record.

Symbol table will be implemented using a structure adhering to the following signature.

```
signature symbolTable =
sig
    type record = {lexeme:string, value:string, Type:string}
    type symTable = record list
    val insertRecord = fn : record -> symTable -> symTable
    val checkIfPresent = fn : string -> symTable -> bool
    val deleteRecord = fn : string -> symTable -> symTable
    val updateType = fn : string -> string -> symTable -> symTable
    val getType = fn : string -> symTable -> string
    val isInitialized = fn : string -> symTable -> bool
end
```

4.2 Implementation

The above mentioned function do as their name suggests. They can be easily implemented using built-in LIST functions and pattern matching on records with few lines of my own code.

5. Semantic Analysis

5.1 Type evaluation during declaration

This section deals with assigning correct types to the variables in their symbol table records. It can be achieved by adding attribute evaluation to the pre-existing parsing algorithms.

The semantic rules used:
Declaration ::= "var" *VariableList* ":" *Type* ";" .
Declaration.type = *Type*.type , *VariableList*.type = *Type*.type
VariableList ::= *Variable* ["," *VariableList*] .
Variable.type = *VariableList*.type

5.1.1 Algorithm

Clearly type behaves like a synthesised attribute. It is passed from parent to child in case of *Variable* and right sibling *Type* to *VariableList*. So I added the type evaluation code after the parsing was completed by traversing the left sibling again.

```
Type (lookahead::tokenList){//Type function must return the type attribute
to its caller function
    switch(lookahead){
    case RES("int",line,ch) : return("int",tokenList);
    case RES("bool",line,ch) : return("bool",tokenList);
    default : errorFn(lookahead,"SyntaxError");
    }
}

reVariableList((ID(id))::(PUN(",",line,chars))::tokens,type,symbolTable){
    newTable = (updateType id type symbolTable)
    (call reVariableList tokens type newTable)
}|reVariableList tokens type table = table

Declaration(tokenList,symTable){//symTable is passed to the parser by Lexer
    val newList = match(tokenList,RES("var",0,0)); //See tokenStructure
    newList = (VariableList newList);
    newList = match(newList,PUN(":",0,0));
    newList,type = (Type newList); //See Type Function
    newList = match(newList,PUN(";",0,0));
    //Parsing Completed now
    altList = match(tokenList,RES("var",0,0));
    newTable = (reVariableList altList, type); /*Updates the symTable with
    appropriate types*/
    return (newList,newTable)
}
```

5.2 Declaration check

The function reVariable can be modified to check if the lexeme id is a valid keyword or not. Validity is checked by Variable Function because it won't accept tokens of the form RES(id,line,chars).

```
reVariableList((ID(id,l,c))::(PUN(",",line,chars))::tokens,type,symbolTable){
    if(!checkIfPresent(id)){
    newTable = (updateType id type symbolTable)
```

```
        (call reVariableList tokens type newTable)}
        else
            errorFn(lookahead,"PreviouslyDeclaredVariable");
    }|reVariableList tokens type table = table
```

5.3 Type Check and Initialization Check at assignments

First if the production rule corresponding to an Assignment is run, the Command function calls IntExpression depending on the ”type” of variable encountered determined previously during variable declaration.

```
    ......
    .inside Command function
        case(ID(id,line,chars){
            if(!(checkIfPresent id symTable))
                errorFn(lookahead,"UndeclardVariable");
            val type = (getType id symTable)
            newList = (Variable lookahead::tokens)
            newList = (match newList PUN("::=",line,chars))
            if(type="bool")
                return ((IntExpression tokens),symTable)
            else
                return ((BoolExpression tokens),symTable)
        }
    ......
```

We also need to add type checking and initialization check in functions of IntExpression and BoolExpression. This check needs to be performed at IntFactor and BoolFactor after parsing has been done. I am showing below for IntFactor.

```
    ....insideIntFactor somewhere
    case(ID(id,line,chars)):
        {
        //before parsing
        if(!(checkIfPresent id symTable))
            errorFn(lookahead,"UndeclaredVariable");
        newlist = (Variable lookahead::tokens)
        //Afterparsing is done
        val type = (getType id symTable)
        val isInit = (isInitialized id symTable)
        if(type!="int")
            errorFn(lookahead,"TypeMismatch");
        elseif(!isInit)
            errorFn(lookahead,"UninitializedVariable");
        else{return (newlist)}
        }
    default:
        //Before raising syntax error check if there’s a type mismatch
        if(lookahead=Bool(val,line,chars)){
            errorFn(lookahead,"TypeMismatch");
        }
        errorFn(lookahead,"SyntaxError");
    .......
```

I'll also add the same code to anotherIntExpression raising the operatorMismatch error if I get BoolOPs instead of arithmetic ones.

6. Three Address Code generation

Three address code generation will take place after parsing and semantic analysis is done. I am using the functions used for parsing itself for this purpose by slightly modifying their code. Each non terminal will emit its code after parsing and if it is an expression it will also return its rvalue to the caller function after parsing and other checks.

6.1 Assignment

```
....inside Command somewhere
case(ID(id,line,chars){
    if(!(checkIfPresent id symTable))
            raise UndeclaredVariable
        val type = (getType id symTable)
        newList = (Variable lookahead::tokens)
        newList = (match newList PUN("::=",line,chars))
        if(type="bool")
            newlist,rVal1 = ((IntExpression tokens),symTable)
        else
            newlist,rVal1 ((BoolExpression tokens),symTable)
    }
    //The above code is the same as parsing except I am saving the rvalues
    and not returning yet.

    ....TAC emission
    emit(id + ' ' + ':=' + rVal1);
......
```

6.2 Conditional Statement

newlabel and newVar return fresh names for label and temporary variable purposes. Also calling a non-terminal will now also emit its code after parsing thus we need to take care of the order in which code is emitted in the caller function.

```
    ....inside Command Somewhere
    case(RES("if",line,chars)){
        newList = match(tokens RES("if",line,chars));
        newList, rValBoolExp = BoolExpression(newList symTable)
        newList = match(newList RES("then",line,chars));
(1).....label=newlabel();
(2).....emit("if" + "(" + rValBoolExp + "==False" + ") goto" + label)
        newList, symTable = CommandSeq(newList symTable)
        newList = match(newList RES("else",line,chars));
(3).....emit(label + ":");
        newList, symTable = CommandSeq(newList symTable)
        newList = match(newList RES("endif",line,chars));
        return (newList,symTable)
    }
    //Marked lines are responsible for 3AC generation.
    //Calling CommandSeq will emit its 3AC.
```

6.3 Iterative Constructs

```
    ...inside Command
    case(RES("while",line,chars):{
        newlist = match(tokens, RES("while",line,chars))
(1).....label1 = newlabel();
(2).....emit(label1 + ":");
        newlist, rValBoolExp = BoolExpression(newlist);
(3).....label2 = newlabel();
(4).....emit("if" + rValBoolExp + "==false goto " + label2)
        newlist = match(newlist, RES("do",line,chars))
        newlist,symTable = CommandSeq(tokenList,symTable)
(5).....emit("goto" + label1);
        newlist = match(newlist, RES("endwh",line,chars));
(6).....emit(label2);
    }........
    \\Numbered code is for 3AC generation, the rest is for parsing
```

6.4 Expression Evaluation

I am showing it only for IntExpression. It will be done in the same manner for all other int and bool parsing functions. Also the anotherIntExpression will return me the operator that it encountered in it's switch case.See section 2.2.

```
    IntExpression (tokenList){
        newlist,rValIntTerm = (IntTerm tokenList));
        newlist,rValIntExp,op = (anotherIntExpression newlist);
(1).....lvalue = newVar();
(2).....emit(rValIntTerm + op + newVar);
(3).....return(newlist,newVar);
    }
```

7. Interpreter

To implement the interpreter I will maintain a list of initialized variables. My interpreter will fetch each instruction line by line. It will then scan all the variables used in that instruction and check if they are initialized. If not it will initialize them first and then process the instruction. Also the inputs to "read" must be passed to the interpreter as a list of strings.

```
\\TAC is the sourcefile of three address code which contains a list of all lines
\\Line is a fn that returns line given the lineNumber
Interpreter(lineNum,inputs,table){
    line = Lexer((Line(lineNum))//To tokenise a line
    switch(line){
        case((ID(id))::tokens):
            newtable = processID(line,table);
            return(Interpreter(lineNum+1),inputs,newtable)
        case((RES("if")::tokens):
            processIF(line,lineNum,inputs,table);//process IF will call Interpreter itself
        case((RES("write")::tokens):
            processWRITE(line,table);
            return(Interpreter(lineNum+1),inputs,table)
        case((RES("read")::tokens):
            newtable,inputs = processREAD(line,table,inputs);
            return(Interpreter(lineNum+1),inputs,newtable)
        case((LABEL(L))::tokens)://Do nothing
            return(Interpreter(lineNum+1),inputs,table)
    }
}
```

7.1 processID

```
processID(line,table){
    newtTable = initialise(id,table) //checks if id is present in table if not
    creates a record for it.
    t1,t2 = fetchTheOperands(line)//gets the operands on the right
    //t2 stores NONE if there is only one operand
    t3,t4 = getValues(t1,t2,table)//fetches the values of t1,t2 from table
    or directly if they are not identifiers.
    op = fetchOP(line)//fetches the operator from the line
    result = process(t3,t4,op)//gives back the result according to op
    newTable = upateVal(id,result,newTable)//Updates the table
    return(newTable);
}
```

7.2 processIF

```
processIF(line,lineNum,inputs,table){
    t1,t2 = fetchTheOperands(line)//operands involved in comparison
    t3,t4 = getValues(t1,t2,table)//fetches the values of t1,t2 from table
    boolOP = fetchOP(line)//fetches the boolOP
    result = compare(op,t3,t4)//computes the result of this operation
    if(result){
        label = fetchLabel(line);
        lineNum = fetchNumber(label,TAC);
        Interpreter(lineNum,inputs,table);//set the interpreter to new line number.
    }else{
        Interpreter(lineNum+1,inputs,table);
    }
}
```

7.3 processWrite

```
processWrite(line,table){
    t1 = fetchOperand(line)
    t2 = getValues(t1,table);
    print(t2);
}
```

7.4 processRead

```
processREAD(line,table,input::inputs){
    id = fetchVariable(line);
    newTable = initialise(id,table) //initialize id if required
    newTable = upateVal(id,result,newTable)
    return(newTable,inputs);
}
```

8. Proof

8.1 Parsing is Correct

The parsing is correct can be confirmed from the grammar. It can be shown that for every non-terminal, given any lookahead(and its type in some cases) we can choose a unique production such that the FIRST(that production)=lookahead. This creates the parse tree without any conflict. The parse tree created is implicit and is determined by the order in which the productions are called. It is created from left to right. Now, to semantically analyse the tree I have just placed the type check code after the parsing code, so that the attributes are analysed from left to right and that of children then the parent.The three address code is emitted using the structure of the implicitly created parse tree. I place the emission of the code in the parent function without disturbing the order of parse function. If the production is that of an expression it emits the its code and then returns the rvalue to its parent function which then used it to print its own 3AC. This means the 3AC of the children is emitted before that of the parent.