Assignment 6: Building a Compiler
CSL 226: Programming Languages

Samarth Singh
2018UCS0065

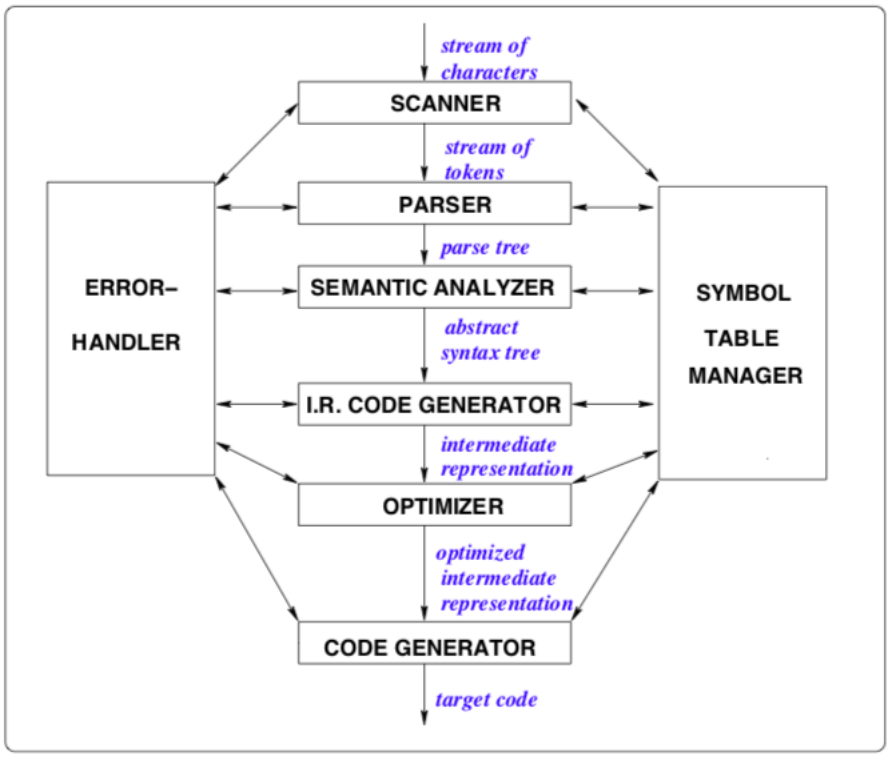
This document provides the Work flow for making a compiler in Standard ML for the WHILE Programming language. The section is divided into four sections explaining the working of each sub-part of the compiler.
The parts of the compiler discussed in the further sections are as follows-

1. Scanner(Lexical Analyser)
2. Parser(Bottom up)
3. IR Code Generator(3 Address opcode format)
4. Interpreter

1 Theoretical Part

1.1 The Big Picture of the Compiler

Before we dive into discussion of building the compiler let us review the flow chart of the sub processes involved using the following flow chart.



The above figure shows the flow of the stream of data, through different parts of the compiler. Starting with the scanner which takes input the code and outputs a stream of tokens.This scanner accounts for the syntactic correction of the code. —-> (*Section* – 2)
The output of the scanner goes through the parser which then converts into a **Parse Tree** along with the construction of a **Symbol Table** storing value of each variable along with its designated scope.—->(*Section* – 3)
The parse tree is further processed using **Syntax directed translation (SDT)** which then traverses the tree and using the attributed grammar of the language converts the tree into **3-Address-code format**.—-> (*Section* – 4)
This IR code generated will then be parsed through the **Interpreter** which will execute the three address codes one by one until the program terminates or it accounts an error.—->(*Section* – 5)

1.2 The WHILE Programming Language

The WHILE Programming language is a simple programming language whose EBNF is given below.

1.3 Tokens of Language WHILE

Reserved Keywords: program, var, int, bool, read, write, if, then, else, endif, while, do, endwh, tt, ff
Integer Operators
. **Unary.**~
. **Binary.** + , -, /, *, %
Boolean Operators
. **Unary.** !
. **Binary.** &&, ||

Relational operators: =, <, >, <=, >=, <>
Assignment Operator: :=
Brackets: (,), {, }
Punctuation: ;, ,
Identifiers: (A-Za-Z)(A-Za-z0-9)*

1.4 EBNF of Language WHILE

$$
\begin{array}{lll}
Program & ::= & \text{“}\mathbf{program}\text{”}\ Identifier\ \text{“::”}\ Block\ . \\
Block & ::= & DeclarationSeq\ CommandSeq\ . \\
DeclarationSeq & ::= & \{Declaration\}\ . \\
Declaration & ::= & \text{“}\mathbf{var}\text{”}\ VariableList\text{“:”}Type\text{“;”}\ . \\
Type & ::= & \text{“}\mathbf{int}\text{”}\ |\ \text{“}\mathbf{bool}\text{”}\ . \\
VariableList & ::= & Variable\text{“,”}\ VariableList\ . \\
CommandSeq & ::= & \text{“}\{\text{”}\{Command\text{“;”}\}\text{“}\}\text{”}\ .
\end{array}
$$

$$
\begin{aligned}
\mathit{Command} \quad & ::= \mathit{Variable}\text{“:=”}\,\mathit{Expression} \mid \\
& \quad \text{“}\mathbf{read}\text{”}\ \mathit{Variable} \mid \\
& \quad \text{“}\mathbf{write}\text{”}\ \mathit{IntExpression} \mid \\
& \quad \text{“}\mathbf{if}\text{”}\ \mathit{BoolExpression} \\
& \quad \text{“}\mathbf{then}\text{”}\mathit{CommandSeq} \\
& \quad \text{“}\mathbf{else}\text{”}\ \mathit{CommandSeq} \\
& \quad \text{“}\mathbf{endif}\text{”} \mid \\
& \quad \text{“}\mathbf{while}\text{”}\ \mathit{BoolExpression}\ \text{“}\mathbf{do}\text{”} \\
& \qquad \mathit{CommandSeq} \\
& \quad \text{“}\mathbf{endwh}\text{”}\ .
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expression} \quad & ::= \mathit{IntExpression} \mid \mathit{BoolExpression}\ . \\
\mathit{IntExpression} \quad & ::= \mathit{IntExpression}\ \mathit{AddOp}\ \mathit{IntTerm} \mid \mathit{IntTerm}\ . \\
\mathit{IntTerm} \quad & ::= \mathit{IntTerm}\ \mathit{MultOp}\ \mathit{IntFactor} \mid \mathit{IntFactor}\ . \\
\mathit{IntFactor} \quad & ::= \mathit{Numeral} \mid \mathit{Variable} \mid \\
& \quad \text{“(”}\mathit{IntExpression}\text{“)”} \mid \text{“~”}\mathit{IntFactor}\ . \\
\mathit{BoolExpression} \quad & ::= \mathit{BoolExpression}\ \text{“||”}\ \mathit{BoolTerm} \mid \mathit{BoolTerm}\ . \\
\mathit{BoolTerm} \quad & ::= \mathit{BoolTerm}\ \text{“\&\&”}\ \mathit{BoolFactor} \mid \mathit{BoolFactor}\ . \\
\mathit{BoolFactor} \quad & ::= \text{“}\mathbf{tt}\text{”} \mid \text{“}\mathbf{ff}\text{”} \mid \mathit{Variable} \mid \mathit{Comparison} \mid \\
& \quad \text{“(”}\mathit{BoolExpression}\text{“)”} \mid \text{“!”}\mathit{BoolFactor}\ . \\
\mathit{Comparison} \quad & ::= \mathit{IntExpression}\ \mathit{RelOp}\ \mathit{IntExpression}\ . \\
\mathit{Variable} \quad & ::= \mathit{Identifier}\ . \\
\mathit{RelOp} \quad & ::= \text{“<”} \mid \text{“<=”} \mid \text{“=”} \mid \text{“>”} \mid \text{“>=”} \mid \text{“<>”}\ . \\
\mathit{AddOp} \quad & ::= \text{“+”} \mid \text{“−”}\ . \\
\mathit{MultOp} \quad & ::= \text{“*”} \mid \text{“/”} \mid \text{“\%”}\ . \\
\mathit{Identifier} \quad & ::= \mathit{Letter}\{\mathit{Letter} \mid \mathit{Digit}\}\ . \\
\mathit{Numeral} \quad & ::= [\text{“+”} \mid \text{“~”}]\mathit{Digit}\{\mathit{Digit}\}\ .
\end{aligned}
$$

The non-terminals Letters and digit are the defined letters (A-Za-z) and (0-9) respectively

2 The Scanner

The Scanner in the compiler making would simply be a **FSM (Finite State machine)**. This State Machine would read character one by one and then will output the stream of tokens to the output file.
The Scanner will detects all the viable tokens and in case of an *syntaxerror* prints an error token. For example an integer token would contain a sequence of integer symbols. The proposed scanner would take input a text file that contains code in WHILE Language and outputs a file containing all the viable tokens.

2.1 The Structure of tokens

Textual representation of the symbols

Symbol	Representation
+,-	"ADDOP"
*,/,%	"MULOP"
~	"NEG"
!	"NOT"
&&	"AND"
||	"OR"
:=	"ASSIGN"
=	"EQUAL"
<>,<,<=,>,>=	"RELOP"
(	"LB"
)	"RB"
{	"LP"
}	"RP"
;	"SC"
,	"COMMA"
program	"PROGRAM"
var	"VAR"
int	"INT"
bool	"BOOL"
if	"IF"
then	"THEN"
else	"ELSE"
endif	"ENDIF"
while	"WHILE"
do	"DO"
endwh	"ENDWH"
read	"READ"
write	"WRITE"
tt,ff	"VBOOL"
(A-Aa-z)(A-Za-z0-9)*	"IDENTF"
(0-9)+	"VINT"

The tokens would be of type **(int)** for tokens except *ADDOP*, *MULOP*, *RELOP*, *VBOOL*, *IDENTF*, *VINT* whose type are described as follows. The **int** field of the above symbols is used to store the line number of the token occurrence.

ADDOP,MULOP,RELOP will be of type **(int*string)** where **int** stores the line number of token and **string** denotes the actual value of the operator.
VBOOL is of type **(int*bool)** where **int** stores the line number and **bool** stores the actual bool value of the symbol which is either true(tt) or false(ff) as described in the above grammar.

VINT is of type **(int*int)** where **int** stores the line number and **int** stores the actual integral value of the sym-

bol.

IDENTF is of type **(int*string)** where **int** stores the line number and **string** representing the actual identifier.

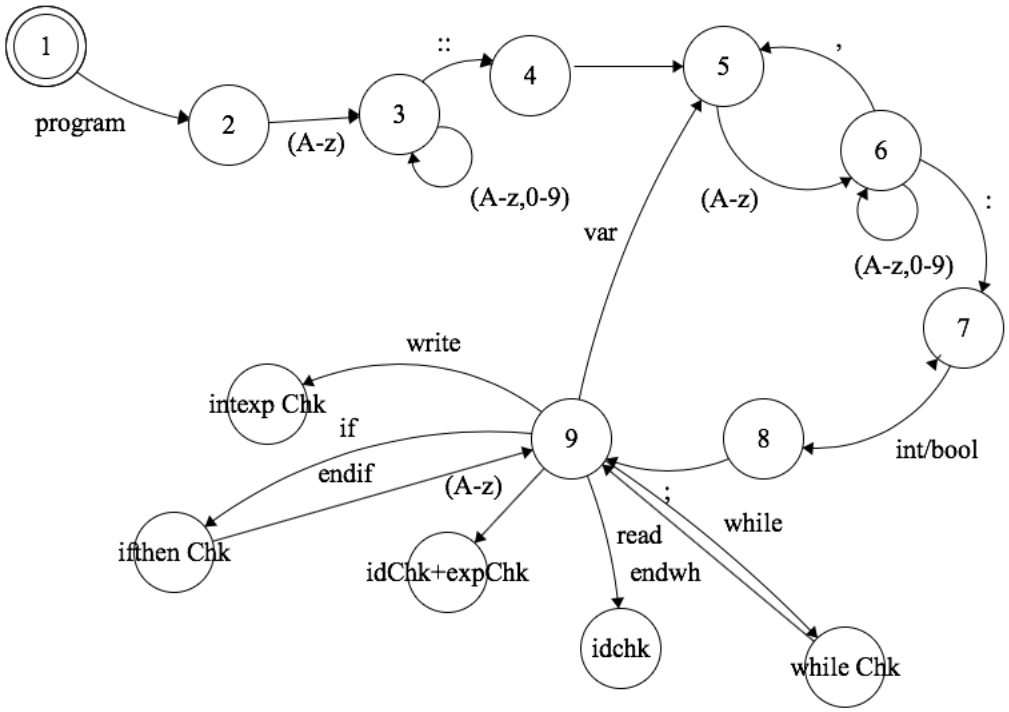
We also define a **ERROR** token of type **(int)** where int stores the line value of the error occuring.

2.2 The Datatype defined using sml

```
datatype TOKEN = ADDOP of int*string
| MULOP of int*string
| NEG of int
| NOT of int | AND of int | OR of int
| ASSIGN of int | EQUAL of int
| RELOP of int*string
| LB of int| RB of int| LP of int| RP of int
| SC of int| COMMA of int| PROGRAM of int| VAR of int
| INT of int| BOOL of bool| IF of int| THEN of int
| ENDIF of int| WHILE of int| DO of int
| ENDWH of int| READ of int| WRITE of int
| VBOOL of int*bool
| IDENTF of int*string
| VINT of int*int
```

2.3 FSM Implementation

The FSM of the scanner consists of accepting and non accepting states. The Finite State Automate will consume symbols and accordingly move to respective states.
The diagram on the next page shows the Automate with different states where we have shorthanded few states into one state for clarity.
In the below shown figure the state *intexp Chk* checks whether the further consuming symbols form a valid integer expression according to the WHILE Grammar.
Similarly, the state *ifthen Chk* check whether the upcoming symbols can form a valid $if\,then else$ expression i.e after **if** there as an valid Boolean expression. Also checking after **then** if there is valid $CommandExp$ etc.
The $idChk + varChk$ checks whether the upcoming symbols can successfully denote an expression of the form $Variable := Expression$; by checking the correctness of firstly an identifier and then after consuming $\varepsilon := \varepsilon$, checking validity of $Expression$.
And lastly, the state $whileCheck$ checks whether the upcoming symbols could successfully denote a statement of form $whileBoolExpdoCommandExpendwh$.Thus checking validity of first the $BoolExp$ and then $CommandExp$ after consuming $while$, do respectively.



3 Parser(Bottom Up)

In this section, we will go through the making of Bottom up Parser. The given grammar is modified by introducing new non-terminals so as to account for productions like-

$CommandSeq \rightarrow \{Command\}$
Here we introduce a new non-terminal let be Y so as to get production rules which could be parsed by a bottom up parser.Thus the new production rule will be-
$CommandSeq \rightarrow CommandY$
$Y \rightarrow CommandY \mid epsilon$

Thus similarly we change a similar occurrence in the EBNF of the WHILE language
$DeclarationSeq \rightarrow \{Declaration\}$
Again by introducing a new terminal X we get the new production rule as
$DeclarationSeq \rightarrow DeclarationX$
$X \rightarrow DeclarationX \mid epsilon$

3.1 Resolving Shift Reduce Conflict

Parsing the above grammar using a bottom up parser would result in **Shift Reduce Conflict** in several cases.
To tackle such Conflicts we apply reduction rules by computing the $FOLLOW$ sets for each such non terminal, one

such example is shown below.
A state transition of the parser makes such transition on consuming the $IntTerm$ non-terminal.

$IntExp \rightarrow IntExp\ AddOp\ \hat{}\ Interm$
$Interm \rightarrow \hat{}\ Interm\ Mulop\ Intfactor$
$Interm \rightarrow \hat{}\ Intfactor$
$Intfactor \rightarrow \hat{}\ Num$
$Intfactor \rightarrow \hat{}\ Var$
$Intfactor \rightarrow \hat{}\ (IntExp)$

On consuming token $IntTerm$ we reach state **S3** given as

$IntExp \rightarrow IntExp\ AddOp\ Interm\ \hat{}$
$Interm \rightarrow Interm\ \hat{}\ Mulop\ Intfactor$

Hence we clearly see that we arrive at a shift reduce conflict where we have a state where reduction and consumption of symbol happen at the same time.
Thus we compute the $FOLLOW$ set for the non-terminal $Intterm$.
Hence $FOLLOW(Intterm) \rightarrow FOLLOW(IntExp) \rightarrow FOLLOW(Expression) \rightarrow$ **";"**
Thus we only apply the reduction rule when the follow up symbol of the next term is **";"**.
Similarly we get another type of shift reduce conflict in the state as shown below-

$Expression \rightarrow IntExp\ \hat{}$
$InExp \rightarrow IntExp\ \hat{}\ Addop\ Intterm$

Here again we have two options of either reducing the $IntExpression$ to $Expression$ or consume an $AddOp$ to get further evaluate the expression.
Again in this case we evaluate the $FOLLOW$ set of the Expression to get $FOLLOW(IntExp)$ as **";"**.
Similarly we get these type of conflicts in $BoolExp$, $BoolFactor$ containing states thus leading to evaluation of $FOLLOW$ sets for each of the following terminal.

3.2 Nonterminals/Terminals and reduction rules

The following are the non-terminals for the bottom up parser.
Non-Terminals: Block, DeclSeq, CommandSeq, Declaration, X, VariableList, Type, Command, IntExp, Interm, Intfactor, BoolExp, Boolterm, Boolfactor, Comparision

The following reduction rules are applied to reduce each terminal to a corresponding Non-terminal by applying appropriate reduction rules given below.

$S \rightarrow$"program" $Identifer$ "::" $Block$ \$ **(Rule 1)**
$Block \rightarrow DeclarationSeq\ CommandSeq$ **(Rule 2)**

$DeclarationSeq \rightarrow DeclarationX$ **(Rule 3)**
$X \rightarrow DeclarationX \mid epsilon$ **(Rule 4)**
$Declaration \rightarrow$"var" $VariableList$ ":" $Type$ ";" **(Rule 5)**
$CommandSeq \rightarrow CommandY$ **(Rule 6)**
$Y \rightarrow CommandY \mid epsilon$ **(Rule 7)**
$Command \rightarrow$"Identifier" ":=" $Expression$ **(Rule 8)**
$Command \rightarrow$"read" "Identifier" **(Rule 9)**
$Command \rightarrow$"write" $IntExpression$ **(Rule 10)**
$Command \rightarrow$"if" $BoolExp$ "then" $CommandExp$ "else" $CommandExp$ "endif" **(Rule 11)**
$Command \rightarrow$"while" $BoolExp$ "do" $CommandExp$ "endwh" **(Rule 12)**
$Expression \rightarrow IntExpression \mid BoolExpression$ **(Rule 13)**
$IntExpression \rightarrow IntExpression$ "AddOp" $Interm \mid Intterm$ **(Rule 14)**
$Intterm \rightarrow Intterm$ "MulOp" $Intfactor \mid Intfactor$ **(Rule 14)**
$Intfactor \rightarrow$"Numeral" | "Identifier" | "(" $IntExpresssion$ ")" | " " $Intfactor$ **(Rule 15)**
$BoolExpression \rightarrow IntExpression$ "||" $Boolerm \mid Boolterm$ **(Rule 16)**
$Boolterm \rightarrow Boolterm$ "&&" $Boolfactor \mid Boolfactor$ **(Rule 17)**
$Boolfactor \rightarrow$"tt" | "ff" | "Indentifier" | $Comparision$ | "(" $BoolExpresssion$ ")" | "!" $Boolfactor$ **(Rule 18)**
$Comparision \rightarrow IntExpression$ "RelOp" $IntExpression$ **(Rule 19)**

NOTE: The $NON - Terminals$ used in the above terms are the tokens declared in the scanner described in Section-2.

3.3 Generation of symbol table

Symbol table is an important part of the compiler. It is used to store information about occurrence of various entities which in this case are the set of $Non - terminals$ and $terminals$ of the grammar.
We will be using the symbol table for the following operations-
Storing the name of each entity in a structured way.
Will be used check if the variable used is declared before it is being called (Declaration before use).
We will be also implementing the table to check whether assignments and expression are **semantically** correct.

3.3.0.1 Structure The Symbol table would be of a list of datatype $ENRTY$ which is declared as-

```
datatype ENRTY = entry of string*string*string
```

Where each string entity will store the values of **symbol name**, **type**, **attribute** respectively.
For eg. for the following expression

```
var salary : int
```

The corresponding entry in the symbol is of type $entry(salary, int, var)$.

3.3.0.2 Operations The following operations would be performed on the symbol table.
insert(): The function will be used to append the entry into the symbol table. The insert function will take a 3-tuple of value, symbol name and attribute as an argument.

For eg. *var x:int*, the compiler should call *insert((x,int,var))*.

lookup(): This function looks for the existence of the symbol in the symbol table, the declaration of the symbol before use, and its initialization.
The argument would be simply the name of the symbol and it would be used as $lookup(symbol)$.

4 Code Generation (3 Address Code)

The Code Generation will be done using Syntax directed translation (SDT) on the reduction rules that are described in $Section - 3$.
Here we associate an attribute with each entity which will help in the translation of code to simpler 3-Addr Code format.
1..**type**
This attribute will be storing the type to which the entity belongs i.e to Int or $Bool$. This will help us in type checking in the production rule
$Command \rightarrow Variable$ ":=" $Expression$
Here in the above production rule we encounter an error if the type of the $Variable$ is of type **int** but the type of $Expression$ is **bool**. Thus we will be wrongly assigning the an "int" type variable to "bool" type expression.
2. **.code**
This attribute will store the corresponding 3-Addr-Code of each entity, these will carry on to the parent of tree Node and will eventually written to the output file.
3. **.place**
This attribute stores the actual value of the variable, this attribute is also carried from the leaves upwards towards the parent Node as each reduction rule is applied.
4. **.begin/ .after**
This attribute stores the starting label and the label just after the end of the code. This attribute will be used to branch and jump to different instructions, in case of commands like **if then else** and **while do**.

4.1 The Semantic rule table

The Semantic actions	
Production Rule	**Semantic Action**
$S \rightarrow$"program" $Identifer$ "::" $Block$ $	insert(identifier.value, program)
$Block \rightarrow DeclarationSeq\ CommandSeq$ $DeclarationSeq \rightarrow \{Declaration\}$ $CommandSeq \rightarrow \{Command\}$	for each $Command$ in $CommandSeq$: emit($Command.code$)
$Declaration \rightarrow$"var" $VariableList$ ":" $Type$ ";" $variableList \rightarrow variableList$ "," $variable$	for each $variable$ in $VariableLsit$: insert($variable.name$,$type.place$,var)
$Command \rightarrow$"Identifier" ":=" $Expression$	chk=lookup(Identifier.place) If chk != nil and $Identifier.type == Expression.type$ then : $Commmand.code$ = (chk = $Expression.place$)

	else throwerror()
$Command \rightarrow$"read" "Identifier"	chk=lookup(Identifier.place) If chk != nil then : $Commmand.code$ = ("read" $Identifier.place$) else throwerror()
$Command \rightarrow$"write" $IntExpression$	$Commmand.code$ = ("write" $IntExpression.place$)
$Command \rightarrow$"if" $BoolExp$ "then" $CommandExp1$ "else" $CommandExp2$ "endif"	$Command.begin$ = newlabel() $Command.after = CommandExp2.after$ $Command.code$ = emit($Command.begin$:) $BoolExp.Code$; emit(if $BoolExp.place = False$ goto $CommandExp2.begin$); $CommandExp1.code$; emit(goto $CommandExp1.after$); $CommandExp2.code$; emit($CommandExp2.after$);

$Command \rightarrow$"while" $BoolExp$ "do"
$CommandExp$ "endwh"
	$Command.begin$ = newlabel()
	$Command.after$ = newlabel();
	$Command.code$ = emit($Command.begin$:); $BoolExp.code$
	emit(if $BoolExp.place = False$ goto $Command.after$);
	$CommandExp.code$; emit(goto $Command.begin$);
	emit($Command.after$);

$Expression \rightarrow IntExpression \mid BoolExpression$	$Expression.place = IntExpression.place$ or $Expression.place = BoolExpression.place$
$IntExpression \rightarrow IntExpression1$ "AddOp" $Interm$	$IntExpression.place$ = newtemp() $IntExpression.code = IntExpression1.code$; $Intterm.code$; emit($IntExpression.place = IntExpression1.code$ $AddOp.value\ Intterm.place$);
$Intterm \rightarrow Intterm1$ "MulOp" $Intfactor$	$Intterm.place$ = newtemp() $Intterm.code = Intterm1.code$; $Intfactor.code$; emit($Intterm.place = Intterm1.code$ $MulOp.value\ Intfactor.place$);
$Intfactor \rightarrow$"Numeral"	$Intfactor.place = Numeral.value$ $Intfactor.code$ = emit($Intfactor.place = Numeral.value$)
.....Similar rules can be applied for BoolExpressions and other Intfactor production rules....	

4.2 The Functions used and definitions

The above table used various functions described below:

emit()

This function appends/emits the address code to the output file or the code accumulator for higher $Non-terminals$.

newtemp()

This functions assigns a new temporary variable/register for the entity. This newly assigned temporary variable can be used further to computer further instructions.

newlabel()
This function assigns a new label for the code. These new labels can then be further used for **jump** and **branch** instructions.
throwerror()
This function writes a error into the output file along with the line number on which it occurs.This error mainly account for semantic as well as type error during assignment.
lookup(),insert() These are the functions that are defined in the section of symbol table.

5 Interpreter

In this section we will be defining the workflow of the interpreter. In the previous three sections we have briefly defined the workflow of the stream of data, on how the processed tokens using a bottom up parser using syntax directed translation can convert to 3-address-code format. Now we will use our interpreter to parse these 3-address instruction and execute to give meaningful output.

5.1 Workflow

The interpreter will have a function $fetchInstruction()$ which take input a line number and return a string containing that whole instruction. The pseudo code for which is given below.

```
fun fetchInstruction(n : int):
    s = open("input_file",r)
    ... code to reach the nth line via loop ..
    instruction =+ s.read() .. until reaches "\n"
    return instruction
```

After getting the instruction we are going to divide it into the following cases-
Case 1:Variable = Constant/ Variable Op Variable
For first case eg. $t1 = 5$ we will assign the same variable using "var" operation in SML.

```
var t5 = 5
```

Similarly for expression *t1 = t2 Op t3* where *Op* is the shorthand for operators $(+, -, *, /, \%)$. We can use the similar declarations of $t2$ and $t3$ have already declared values.

Case:2 if cond goto label
In this case we evaluate the condition using again a "var" variable, then by using conditional statements "if then else" in SML we decide to branch to some instruction or not.The below is the pseudo code showing the implementation.

```
var chk = cond ... evaluate depending whether it is
               ... a relational statement or a boolean
               ... expression
```

```
if chk then return fetch_instruction(jumplabel(label))
else fetch_instruction(x+1)
```

In the above implementation **jumplabel()** is a function taking argument a label and returns the line number of the instruction.
x is the global variable keeping the track of the current pointer. The pseudo code for *jumplabel* function is given below.

```
fun jumplabel(label: string):
    s= open('inputfile.txt', read)
    current_label = read_label(s) .. function to read
                                  .. the current label
    var line = 1
    if(current_label != label) then
                recfunc(s+1,label,line+1)
    else line
```

Here **recfunc** is a recursive function that will be declared inside the **jumplabel** function to recursively call similar function with parsing point of file to next line and the line incremented by one.

Case 3: goto label
Similar evaluation of this instruction will be done as discussed above using **jumplabel** function. The only difference is that now there is no need to check the condition since this is an unconditional branch.

Case 4: read variable
We will again use "var" for declaring the variable.The pseudo code is shown below.

```
var t6 = read.console(a:int);
```

Case 5: write expression
In this case we only need to call a function *writetoFile*, the pseudo code implementation is given below.

```
fun writetoFile(a:int) :
    s = open("outputfile.txt",w)
    s.append(a)          ... append function to write at
                         ... end of the file
```

6 Concluding Remarks

The following documentation is does not describe the exact functions used in the actual implementation. The reader must also note that this design documentation is subjected to change during the course of further implementation.

7 Sources

The following sources were used in making of the documentation.

1. Class slides (http://www.cse.iitd.ac.in/ sak/courses/pl/pl.pdf)
2. Sethi R: Programming Languages: Concepts and Constructs, 2nd ed., Addison-Wesley, 2007.
3. Modern Compiler Implementation in ML ©1998 by Andrew W. Appel Published by Cambridge University Press