# 1 Introduction

As given in Assignment 5, Figure 1 is the overall structure of program execution. In this assignment(A6), we will be generating code from the Abstract Syntax Tree generated in Assignment 3. This assignment has three steps:

- Semantic Analysis : For this, you will have to do a tree walk over the AST. This step includes connecting variable definitions to their uses, type checking and translating the abstract syntax tree into a simpler representation suitable for generating machine code. This step would require the Abstract Syntax Tree generated in A3 along with the symbol table generated in A2 as input.
- Code Generation: Once you have modified the abstract syntax tree, you will use it to generate the low level code described in A5.
- Integrating Assignments: This assignment will be integrated with A4 and A5.

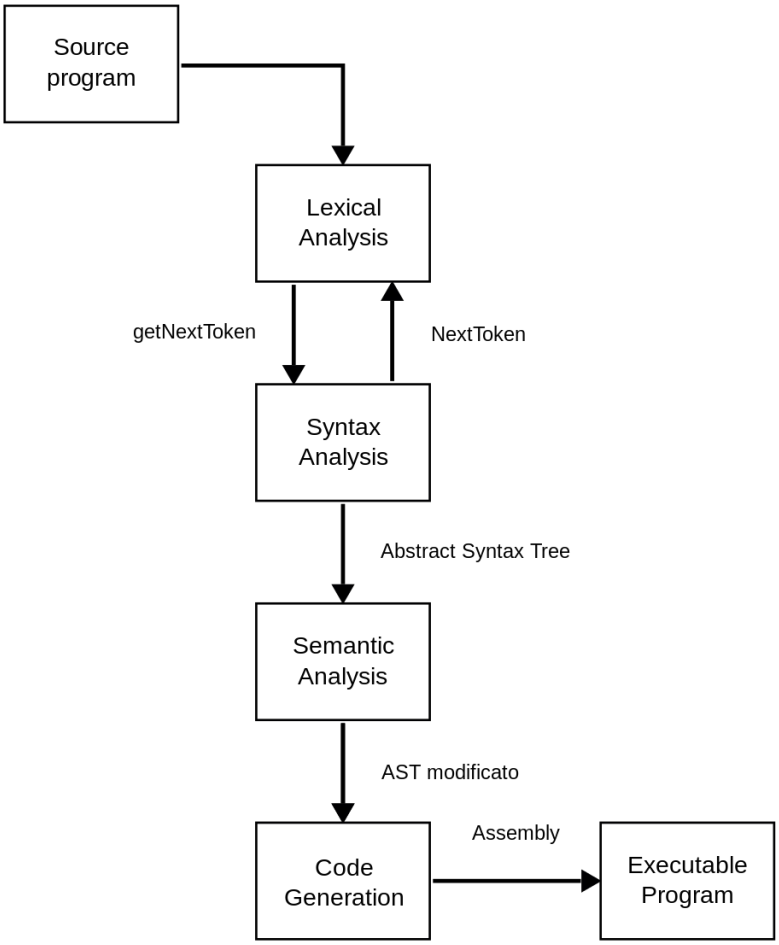


Figure 1: A multipass compiler

# 2 Semantic Analysis

## 2.1 Symbol Table

As you perform semantic analysis on the AST or generate code, you will need to maintain a Symbol table in the program. A good idea is to maintain the symbol table as a tree and walk it as you walk the AST.

## 2.2 Declaration of Variables

Consider the following program

```
int a;
proc foo
bool c;
{
c := a > b;
};
{
}
```

When you are checking for declaration of a variable, you first check for the declaration within the procedure (You get a positive match for $c$ but a negative match for $a, b$. Then, you check for declaration at a higher scope (global in this case) and you get a positive match for $a$. Since you cannot go up anymore, you will conclude that variable $b$ has not been declared.

## 2.3 Using Local and Global Variables

Now, consider the following program:

```
int a, b;
bool c;
proc foo
bool c;
{
c := a > b;
};
{
c := b > 4;
}
```

In this program, you have a boolean variable declaration $c$ both a global level as well as the procedure level. When you are executing line 6 of the program, you will use the local variable. However, when you are executing line 9 of the program, you use the global variable.

## 2.4 Type Checking

The Binary and Unary Operators for integer and boolean variables were defined in A1. The relational operators were defined for both integers and boolean. However, you cannot mix the types. For example:

```
a < b
```

For this to be correct, either both $a$ and $b$ should be either integers or bools. You cannot have an operator between an integer and boolean. e.g. $5 < tt$ is not allowed.

# 3 Generating Code

## 3.1 Memory Array

While converting the code into 3 address code, you'll be allocating memory to variables to store their values and need a memory array. Since the program has arbitrary length integers, instead of directly saving the variables in the memory array, it can simply contain references to the heap memory where the actual values are stored. The name of the variable will be an index to this memory array. This will ensure that each element of the memory array is of fixed length. Values of boolean variables could also be references for uniformity or they could be stored in the assigned location itself.

# 4 Low Level Code

The assignment will generate a 3 address code with the following instruction set. The names have been kept mnemonic but description has also been provided with it:

```
DECLARE_INT ident _ _ (declares int variable)
DECLARE_BOOL ident _ _ (declares a boolean variable)
DECLARE_PROC ident pc _ (declare procedure name and store the program counter
of the first command in the proc in pc)
PRINT ident _ _ (prints the value of Identfier)
READ _ _ ident(reads the value from stream into ident)
CALL procname _ _ (and store the return address)
IF ident location _ (jump to location if ident is false)
GOTO _ location _ (jump to location)
RETURN _ _ _ (jump to the return address)
ASSIGN ident1 _ ident2 (Store the value of ident1 in ident 2)
OPER_BINARY id1 id2 id3  (id3 <- id1 OPER_BINARY id2 for all binary operators)
OPER_UNARY id1 _ id3 (id3 <- OPER_UNARY id1 for all unary operators)
END_OF_CODE _ _ _ ( Marks the end of the program)
```

Here OPER_BINARY $\in \{\mathsf{PLUS}, \mathsf{MINUS}, \mathsf{MULT}, \mathsf{DIV}, \mathsf{MOD}, \mathsf{GEQ}, \mathsf{GT}, \mathsf{LEQ}, \mathsf{LT}, \mathsf{NEQ}, \mathsf{EQ}, \mathsf{AND}, \mathsf{OR}\}$ is a binary operator; OPER_UNARY $\in \{\mathsf{UMINUS}, \mathsf{NOT}\}$ is a unary operator.

Note that while generating the code, you might have to generate temporary variables. Consider the following example:

```
int a, b, c;
{
    b := 5;
    c := 2;
    a := b*c;
    print(a);
}
```

The machine code for the above example would be:

```
DECLARE_INT a _ _
DECLARE_INT b _ _
DECLARE_INT c _ _
ASSIGN 5 _ b
ASSIGN 2 _ c
MULT b c a
```

```
PRINT a _ _
END_OF_CODE _ _ _
```

The output for the program would be:

```
10
```

### 4.1 Generating Fresh Variables

It might happen that while generating the code, you need to generate and allocate memory to new variables. Each of these new variables should have an entry in the symbol table as well. Note that the symbol table contains all the context-sensitive information for the language.

## 5 Integration with Other Assignments

This is NOT a standalone assignment. You will need to use the BigInt Package you programmed in A4 and then evaluate the instructions you generated using the VM in A5.

## 6 I/O Specifications

- **Input:** The input to the program would be an Abstract Syntax Tree (output of A3) and the symbol table generated in A2.
- **Output:** The output of the program would be two files. The first one will contain the instructions to be used in A5. The second one would contain the output of the program.

## 7 Instructions for Submission

- All submissions must be through moodle. No other form of submission will be entertained.
- No submissions will be entertained after the submission portal closes.
- Sometimes there are two deadlines possible – the early submission deadline (which we may call the "lifeline") and the final "deadline". All submissions between the "lifeline" and the "deadline" will suffer a penalty to be determined appropriately.

## 8 What to Submit?

- You will create one folder which will have three program files and the writeup file.
- The program file for A6 should be named with your Kerberos ID. For example, if kerberos id is 'cs1140999' then the file name should be cs1140999.sml or cs1140999.ocaml. The writeup should be named as "writeup.txt". The other files can have any names you like as they would be used inside your program and not explicitly run.
- All the files should be present in one folder. Your folder also should be named as your Kerberos ID. For example, if kerberos id is 'cs1140999' then the folder should be called cs1140999.
- The first line of writeup should contain a numeral indicating language preferred with 0-ocaml, 1-sml and 2-haskell.
- For submission, the folder containing the files should be zipped(".zip" format). Note that, you have to zip folder and NOT the files.

- This zip file also should have name as your Kerberos ID. For example, if kerberos id is 'cs1140999' then the zip file should be called cs1140999.zip.
- Since the folder has to be zipped the file cs1140999.zip should actually produce a new folder cs1140999 with your program files and writeup.txt.

- After creating zip, you have to convert ".zip" to base64(.b64) format as follows (for example in ubuntu): *base64 cs1140999.zip > cs1140999.zip.b64* will convert .zip to .zip.b64
  This cs1140999.zip.b64 needs to be uploaded on moodle.
- After uploading, please check your submission is up-to the mark or not, by clicking on evaluate. It will show result of evaluation. If folder is as required, there will be no error, else REJECTED with reason will be shown. So, make sure that submission is not rejected.