Assignment 4

Problem Statement: Implementation of Semantic Analysis Operations (like type checking, verification of function parameters, variable declarations and coercions) possibly using an Attributed Translation Grammar.

Objective: To understand semantic analysis operations

Outcome: Implementation of semantic analysis for varible declaration using lex and yacc

Software Requirements: LEX,YACC,GCC

Hardware Requirements: 4GB RAM, 500 GB HDD

Theory:

Semantics:

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Functions of Semantic Analysis:

1. **Type Checking –**
   Ensures that data types are used in a way consistent with their definition.
2. **Label Checking –**
   A program should contain labels references.

3. **Flow Control Check –**
   Keeps a check that control structures are used in a proper manner.(example: no break statement outside a loop)

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

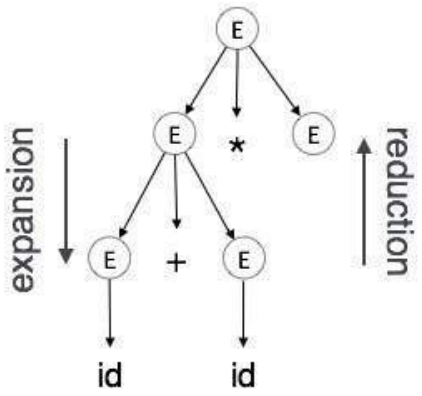
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

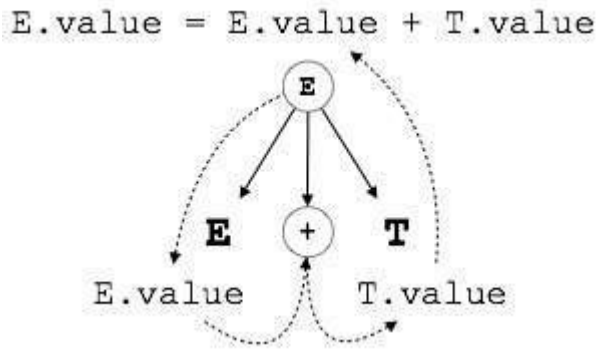
For example:

int value = 5;
<type, “integer”>
<presentvalue, “5”>

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

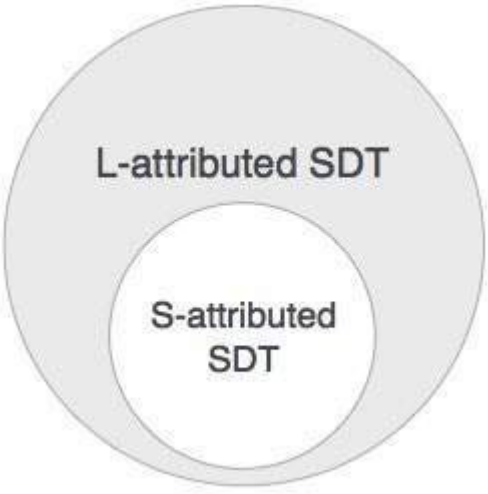
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

S → ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Conclusion:
Thus we've implemented semantic analysis for variable declaration using lex and yacc tool.

Code:

var.l-

```
%{
#include<stdio.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
void|int|float|char return BUILTIN;
, return COMMA;
; return SC;
"\n" return 0;
")" return RP;
"(" return LP;
[a-zA-Z0-9]* return ID;
%%

int yywrap(void) {
return 1;
}
```

var.y-

```
%{
#include<stdio.h>
int yylex(void);
void yyerror(char *);
int flag=0;
%}
%token COMMA ID BUILTIN SC RP LP
%%
var:BUILTIN EXP SC {flag=1;}
;
EXP:EXP COMMA ID| ID |ID LP BUILTIN ID RP {flag =1;}
;
%%


int main()
{
```

```
yyparse();
if(flag==1)
{
printf("valid Declaration\n");
}
else
{ printf("Invalid Declaration\n");
}
return 0;
}


void yyerror(char *s)
{
printf(" ");
}
```

Output-

```
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment4$ lex var.l
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment4$ yacc -d
var.y
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment4$ gcc
y.tab.c lex.yy.c -ll -ly -lm
(base) dell@dell-Inspiron-15-3567:~/BEALL2/Compiler/Compiler/Assignment4$ ./a.out
int a;
 valid Declaration
```