University of Mosul
College Computer Science & Mathematics
Department of Computer Science

Compiler 2

Lecture 3: Syntax Directed Translation

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax-directed translation rules use:

1. Lexical values of nodes.
2. Constants.
3. Attributes associated with the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

Example

E → E+T | T
T → T*F | F
F → id

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example, we will focus on the evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

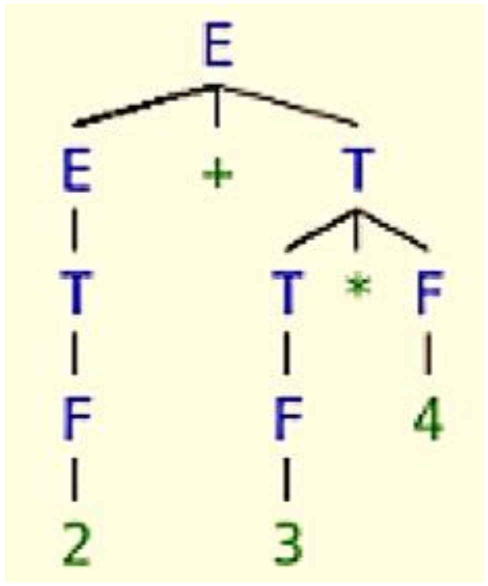
1. **E →**	**E + T**	{ E.val = E.val + T.val }
2. **E →**	**T**	{ E.val = T.val }
3. **T →**	**T * F**	{ T.val = T.val * F.val }
4. **T →**	**F**	{ T.val = F.val }
5. **F →**	**id**	{ F.val = id.lexval }

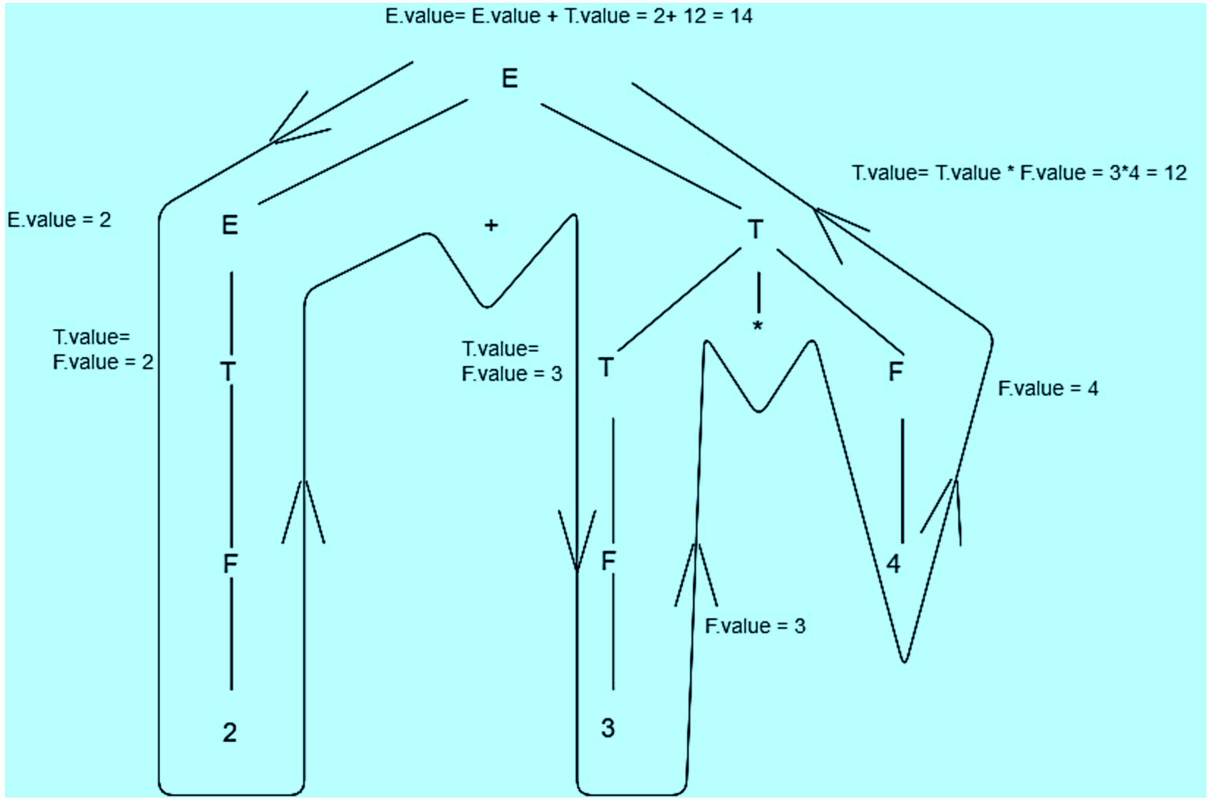
Semantic analysis for (S = 2+3*4)

To evaluate translation rules, we can employ one depth-first search traversal on the parse tree. This is possible only because SDT rules don’t impose any specific order on evaluation until children’s attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom-up in the left to right fashion for computing the translation rules of our example.

Example

E → E+T | T
T → T*F | F
F → id





S–attributed and L–attributed SDTs in Syntax directed translation

Attributes may be of two types – Synthesized or Inherited.

1- Synthesized attributes

A Synthesized attribute is an attribute of the non-terminal on the ***left-hand side*** of a production. Synthesized attributes represent information that is being passed up the parse tree. ***The attribute can take value only from its children*** (Variables in the RHS of the production).

For eg. let’s say A → BC is a production of a grammar, and A’s attribute is dependent on B’s attributes or C’s attributes then it will be synthesized attribute.

2- Inherited attributes

An attribute of a nonterminal on the ***right-hand side*** of a production is called an inherited attribute. The attribute can take value either from its ***parent or from its siblings*** (variables in the LHS or RHS of the production).

For example, let’s say A → BC is a production of a grammar and B’s attribute is dependent on A’s attributes or C’s attributes then it will be inherited attribute.

S-attributed SDT:

If an SDT uses only synthesized attributes, it is called as S-attributed SDT.

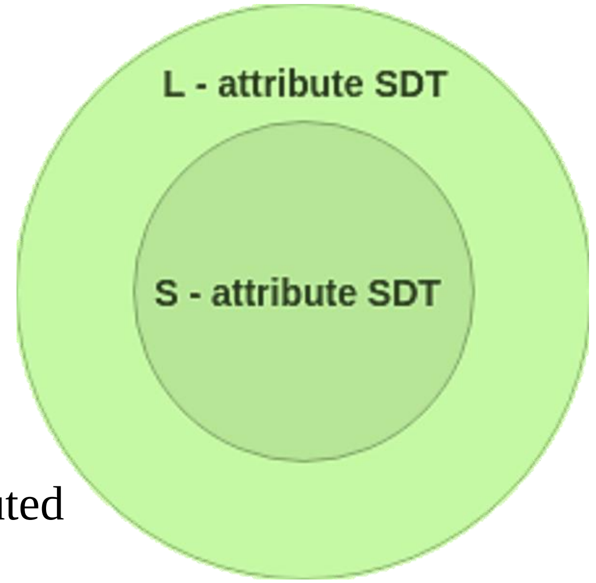
S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

Semantic actions are placed in rightmost place of RHS.

L-attributed SDT:

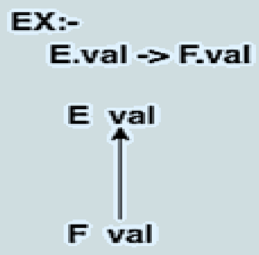
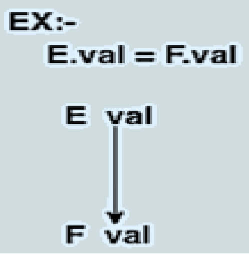
If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Semantic actions are placed anywhere in RHS.



For example: A $\rightarrow$ XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.
Note – If a definition is S-attributed, then it is also L-attributed but NOT vice-versa.

The comparison between these two attributes are given below:

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed STD.	Inherited attribute is used by only L-attributed SDT.
7.		

University of Mosul
College Computer Science & Mathematics
Department of Computer Science

Compiler 2

Lecture 4: Semantic Analysis

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Semantic Analysis in Compiler Design

Semantic Analysis is the third phase of compilation. Semantic analysis makes sure that declarations and statements in programs are semantically correct. It is a collection of procedures that are called by the parser as and when required by grammar. Both the syntax tree of the previous phase and the symbol table are used to check the consistency of the given code. **Type checking** is an important part of semantic analysis, where the compiler makes sure *that each operator has matching operands*.

Semantic Analyzer:

It uses the syntax tree and symbol table to check whether the given program is semantically consistent with the language definition. It gathers type information and stores it in either the syntax tree or symbol table. This type of information is subsequently used by the compiler during intermediate-code generation.

Errors recognized by semantic analyzer are as follows:

1. Type mismatch
2. Undeclared variables
3. Reserved identifier misuse
4. Multiple declaration of variable in a scope.
5. Accessing an out-of-scope variable.
6. Actual and formal parameter mismatch.

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)

Example:

float x = 10.1;

float y = x*30;

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

Static and Dynamic Semantics:

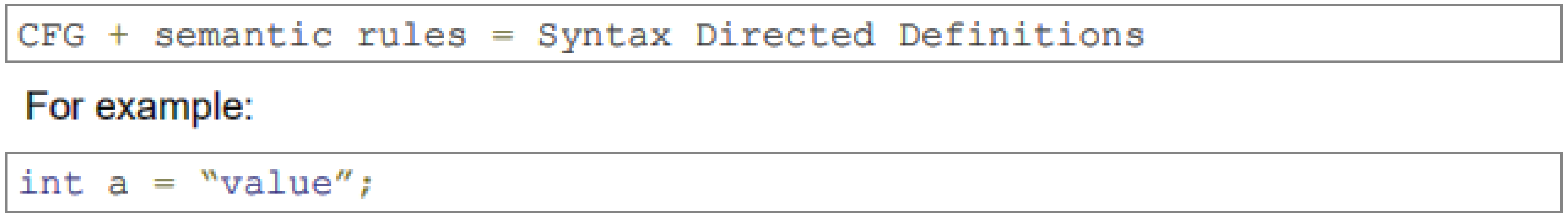
In many compilers, the work of the semantic analyzer takes the form of semantic action routines, invoked by the parser when it realizes that it has reached a particular point within a grammar rule.

Of course, not all semantic rules can be checked at compile time. Those that can are referred to as the static semantics of the language. Those that must be checked at run time are referred to as the dynamic semantics of the language. C has very little in the way of dynamic checks.

Examples of rules that other languages enforce at runtime include the following:

■ Variables are never used in an expression unless they have been given a value.
■ Pointers are never dereferenced unless they refer to a valid object.
■ Array subscript expressions lie within the bounds of the array.
■ Arithmetic operations do not overflow.

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 the 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

If a semantic analyzer has a symbol table for each separate procedure, it can find semantic errors that occur because of the following mistakes:

- Names that aren't declared
- Operands of the wrong type for the operator they're used with
- Values that have the wrong type for the name to which they're assigned

If a semantic analyzer has a symbol table for the program as a whole, it can find semantic errors that occur because of the following mistakes:

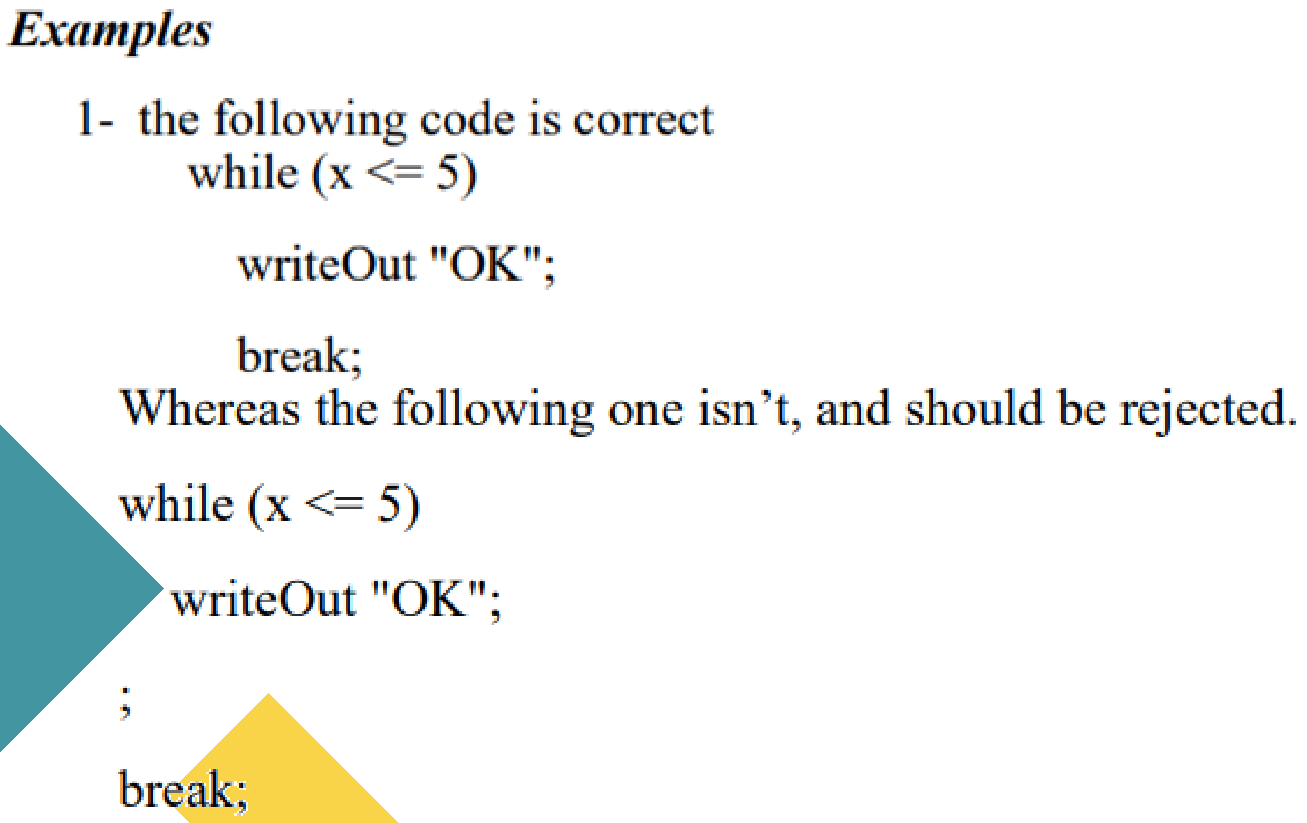
- Procedures that are invoked with the wrong number of arguments
- Procedures that are invoked with the wrong type of arguments
- Function return values that are the wrong type for the context in which they're used

If a semantic analyzer has control-flow and data-flow information for each separate procedure, it can find semantic errors that occur because of the following mistakes:

- Code blocks that are unreachable
- Code blocks that have no effect
- Local variables that are used before being initialized or assigned
- Local variables that are initialized or assigned but not used

If a semantic analyzer has control-flow and data-flow information for the program as a whole, it can find semantic errors that occur because of the following mistakes:

- Procedures that are never invoked
- Procedures that have no effect
- Global variables that are used before being initialized or assigned
- Global variables that are initialized or assigned, but not used



Examples

1- the following code is correct

```
    while (x <= 5)

        writeOut "OK";

        break;
```

Whereas the following one isn't, and should be rejected.

```
while (x <= 5)

  writeOut "OK";

;

break;
```

2-

```
x = 3;
 z = "abc";
y = x + z;
```

The three lines above should also generate a compilation error. The reason is that the operator + is used with a int type (x) and a string type (z). Even though this kind of operation may be allowed in some languages.