Compiler Construction

Rationale

Language translation techniques play a vital role in many fields of Computer Technology. The need for language translators arises because the language in which one wants to express ones computational needs is significantly different from the language recognised by a real machine. Development of translators, therefore, have resulted in two important fallouts. On one hand it has a strong influence on the design of new programming languages, on the other the issue of efficient translation of programming languages is inherently linked with the design of new architectures for higher throughputs. Language translation also plays an important role in the design of command languages and user interfaces of software systems, which require knowledge of scanning and parsing techniques.

In the current context, it is essential for computer science undergraduate students to develop a thorough knowledge of the theory and techniques of language translation. This course is aimed at imparting this knowledge through a detailed treatment of the theory of language translation, and a familiarity with the use of currently available software tools for automatic generation of various modules of a language translator.

Objectives

At the end of this course on Compiler Construction, the student should be able to

1. Understand the two models of language processing, viz. compilers and interpreters, and realize their strengths and limitations.

2. Understand the influence of various features of programming languages and machine architectures on the complexity and efficiency of language translation.

3. Appreciate the role of lexical analysis in compilation, understand the theory behind design of lexical analysers and lexical analyser generators, and be able to use Lex to generate lexical analysers.

4. Be proficient in writing grammars to specify syntax, understand the theories behind different parsing strategies – their strengths and limitations, understand how generation of parsers can be automated, and be able to use Yacc to generate parsers.

5. Understand the error detection capabilities of a compiler and how to report them in a user friendly manner.

6. Understand the need for various static semantic analyses such as declaration processing, type checking, and intermediate code generation, and be able to perform such analysis in a syntax directed fashion through the use of attributed definitions.

7. Understand the issues involved in allocation of memory to data objects, and the relation of such issues with higher level language features such as data types, scope and nesting rules, and recursion.

8. Understand the key issues in the generation of efficient code for a given architecture.

9. Understand the role played by code optimisation.

Compiler Construction

Course Modules

Module 1 : Introduction to Language Processing

Module 2 : Lexical Analysis

Module 3 : Syntax Analysis

Module 4 : Error Analysis

Module 5 : Static Semantics and Intermediate Code Generation

Module 6 : Run-time Environments

Module 7 : Code Generation

Module 8 : Code Optimisation

Preface

In this module, we shall first discuss the issues in static semantic analysis and how to perform such analyses in a syntax directed fashion by augmenting the context free grammar with semantic rules and the parsing process by attribute evaluation. We shall then study S-attributed schemes and see how the evaluation of attributes in a S-attributed scheme can be integrated with parsing. Next, we shall study the issues involved in performing semantic analysis such as declaration processing, type checking, and intermediate code generation for various language constructs. Finally, we shall discuss L-attributed schemes, their advantage over S-attributed schemes and the evaluation of attributes in L-attributed schemes.

Contents

1 Introduction

Static semantic analysis ensures that a program is correct, in that it conforms to certain extra-syntactic rules. The analysis is called

static because it can be done by examining the program text, and

semantic because the properties analysed are beyond syntax, i.e. they cannot be captured by context free grammars.

Some examples of such analyses are (i) type analysis, (ii) name and scope analysis, and (iii) declaration processing. Further, the techniques used for static semantic analysis can also be used for intermediate code generation. Therefore, though generation of intermediate code is a synthesis activity, we shall study it with the analysis mentioned above.

1.1 Type analysis

Type analysis ensures that the type of an object matches that expected by its context.Examples of issues issues involved in type analysis are:

1. Is the variable on the left hand side of an assignment statement 'compatible' with the expression on the right hand side?
2. Is the number of actual parameters and their types in a procedure call compatible with the formal parameters of the procedure declaration?
3. Are the operands of a built-in operator of the right type? If not, can they be forced (coerced) to be so? What is the resultant operator, then, to be interpreted as? As an example, the expression $3 + 4.5$ can be interpreted as integer addition of 3 and 4 (4.5 truncated), or real addition of 3.0 (3 coerced to real) and 4.5.

Note that the notion of 'type compatibility' is non-trivial. For example, in the Pascal declaration

```
type ptr = ^ object;
var a :  ^ object; b :  ptr;
```

are `a` and `b` type compatible? Different languages give different answers to this question.

1.2 Name and scope analysis

In a language such as Pascal, the same name (identifier) may appear in more than one declarations. So, given the use of a name at a program point, it is natural to ask whether the use is valid at that point, and if so, which declaration does it correspond to. This is because properties such as type get bound to a name in a declaration. Conversely, we can also ask from which regions of a program is a name in a declaration visible. Such questions are answered by name and scope analysis.

As an example, consider the following program outline with a non-trivial nesting structure.

```
program S;
   var a,x:  _
   procedure R;
      var i:  _
      procedure T;
         var m,n:  _
         body of T;
      body of R;
   procedure E;
      body of E;
   procedure Q;
      var k,v:  _
      function P(y,z:_):_
         var i,j:  _
         body of P;
      body of Q;
   body of S.
```

The names visible from the body of `P` are `y, z, i, j, k, v, a, x, P, Q, E, R and S`. Name and scope analysis would ensure that only the names listed above would be refered to from `P`. The procedure `P` is visible from `P` itself and `Q`.

Note that by name we mean names of variables, types, functions and even labels. In Pascal, for example, the following is not allowed:

```
begin
   ...
```

```
        go to 10;
        ...
        while E do
           begin
           ...
           10:  ...
           end
        ...
     end;
```

Here, label 10 is the name. It is only visible in the block in which it is defined.

1.3 Declaration processing

Declaration processing includes several activities. Firstly, uniqueness checks ensure that an identifier is declared uniquely in an declaration. Then, as declarations provide most of the information regarding properties of a name, this information must be recorded in a data structure called the *symbol table*. The main issues here are (i) what information must be entered into the symbol table and how should such information be represented, and (ii) what should be the organization of the symbol table itself. Moreover, as declarations are processed, the addresses of variables may also be computed. This address is in the form of an offset relative to the beginning of the storage allocated for a block (refer to the module on runtime environments for more details).

1.4 Intermediate code generation

Many compilers are organized as a combination of a *front end* and a *back end*. The front end translates the input program into an intermediate representation, from which the back end generates the target code. This has several advantages. The intermediate representation enhances the possibility of better code generation and code optimization. This organization also aids retargetability. To generate a new compiler for the same source language but a different target, one has to rewrite the back end only.

As we shall see, there is is a choice of intermediate representation from many possible alternatives such as abstract syntax trees, postfix code and three address code. As an example, the three address code generated for the assignment statement `x := A[y,z] < b`, where `A` is a 10×20 array of integers, and `b` and `x` are reals is:

```
1.  t1 := y * 20
2.  t1 := t1 + z
3.  t2 := addr(A) - 21
4.  t3 := t2[t1]
5.  t4 := intoreal(t3)
6.  if t4 < b goto 9
7.  t5 := 0
8.  goto 10
9.  t5 := 1
10.  x:= t5
```

The name three address code comes from the fact that most of the instructions are of the form $result := operand_1 \ operator \ operand_2$ and therefore involve three addresses, the addresses of the two operands and the address of the result. Since an instruction in three address code can have only a single operator on the right hand side of the assignment, the compiler generates temporary variable names like `t1`, `t2` etc. to store results of intermediate computations.

2 Syntax directed analysis

Can we always represent the properties stated above through a context free grammar? For instance, can we write a context free grammar capturing only those programs in which the number of actual parameters in a procedure call matches that of formal parameters in a procedure declaration. There are theoretical results to show that the answer is in the negative. For example, consider The language $\{a^n b^m c^n d^m | n \geq 1 \ and \ m \geq 1\}$ is an abstraction of languages with two procedures and their corresponding calls, which require that the number of formal parameters (represented by the as and bs) be the same as the number of actual parameters (cs and ds). It can be proven that this language is not context free. Similarly, the language $\{wcw| \ w \ is \ a \ string \ from \ some \ alphabet \ \Sigma\}$ can be thought of as an abstraction of languages with a single identifier declaration (the first occurrence of

w), which must be declared before use (the second occurrence of w). This language is also not context free.

However, we would like to do static analysis in a ***syntax directed*** fashion, i.e. along with the process of parsing. This is done by adding extra machinery to context free grammars and the process of parsing.

2.1 Syntax directed definition

We associate with each terminal and non-terminal a set of values called ***attributes***. The evaluation of attributes takes place during parsing., i.e. when we use a production of the form $A \rightarrow XYZ$ for derivation (in the case of top-down parsers) or reduction (in bottom-up parsers), the attributes of X, Y and Z could be used to calculate the attributes of A.

The attributes chosen depend on the type of analysis that is required to be performed. As an example, suppose we wanted to generate intermediate code for the grammar shown below:

$$
\begin{aligned}
&S \rightarrow id := E \\
&E \rightarrow E_1 + E_2 \\
&E \rightarrow E_1 * E_2 \\
&E \rightarrow -E_1 \\
&E \rightarrow id
\end{aligned}
$$

Then we shall choose the following attributes:

$S.code$ – The intermediate code associated with S.
$E.code$ – The intermediate code associated with E.
$E.place$ – The temporary variable which holds the value of E.
$id.place$ – The lexeme corresponding to the token id.

Now the grammar is augmented with the following semantic rules:

$$
\begin{array}{ll}
S \rightarrow id := E & S.code := E.code \parallel gen(id.place\ ' :=' \ E.place); \\
E \rightarrow E_1 + E_2 & E.place := newtemp; \\
 & E.code \ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\qquad ' :=' \ \ E_1.place\ ' +' \ \ E_2.place); \\
E \rightarrow E_1 * E_2 & E.place := newtemp; \\
 & E.code \ := E_1.code \parallel E_2.code \parallel gen(E.place
\end{array}
$$

$$' :=' \ \ E_1.place \ ' *' \ \ E_2.place);$$

$E \to -E_1$ $\qquad E.place := newtemp;$

$\qquad\qquad E.code \ := E_1.code \parallel gen(E.place \ ' :=' \ \ 'uminus' E_1.place);$

$E \to id$ $\qquad E.place := id.place;$

$\qquad\qquad E.code \ := \ ' \ ';$

We shall first explain the notations used in the above definition. *newtemp* is a function which generates a new temporary variable name each time it is invoked, and '$\parallel$' is the concatenation operator. Since the intermediate code is essentially a string, we form larger fragments of code by concatenating smaller ones. *gen* evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string. Also observe that the nonterminals E_1 and E_2 are the same as the nonterminal E. The subscripts are used to distingiush between different occurrences of a nonterminal.

The first semantic rule is to be read as: the code for S is the code for E followed by a single assignment. This assignment statement has the name of the identifier id on the left hand side and the name of the temporary variable which holds the value of E in the code for E.

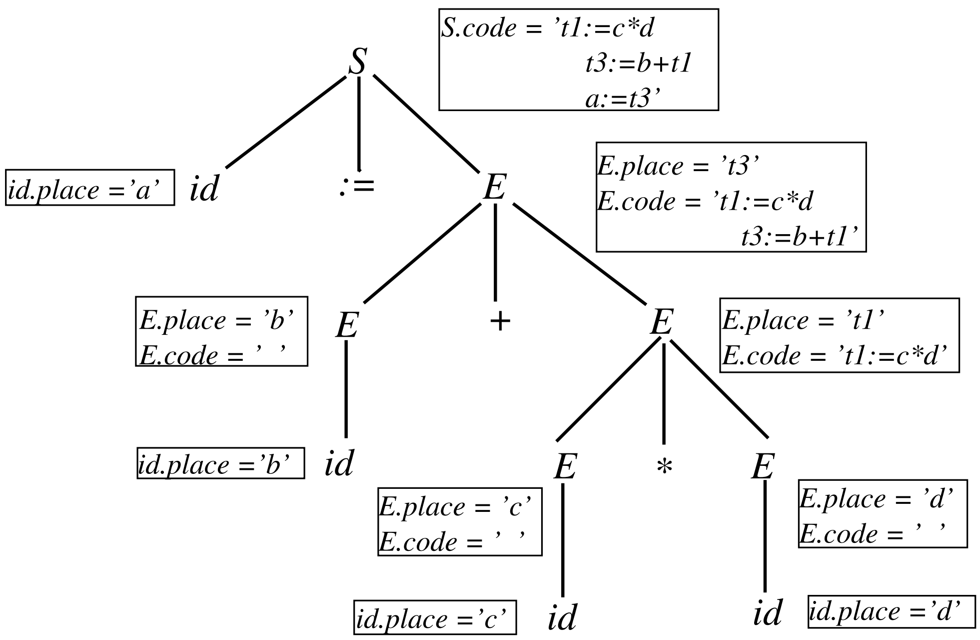


Figure 1: Parse tree and attributes

Let us explain the above with an example. The attributes of the various nonterminals for the statement `a := b + c * d` are shown in figure 1. Consider the nonterminal E corresponding to the expression `b + c * d` in the parse tree. The intermediate code for E, as shown in the figure is `t1 := c * d; t3 := b + t1`. This code, when evaluated, evaluates the values of E in the temporary variable `t3`. Therefore $E.place$ is `t3`. Now consider the region of the parse tree where the production $S \to id := E$ is being used. The associated semantic rule gives the code for S as `t1 := c * d; t3 := b + t1; a := t3`.

A syntax directed definition is a kind of abstract specification. What a semantic rule merely specifies is the way in which attributes of grammar symbols are related to each other. They do not specify when the attributes are to be computed, or the data structures which would hold attribute values. As we shall see later, attribute should proceed along with the parsing process. This would introduce restrictions on the possible relations between attribute values. Also note that the attributes of terminals are usually supplied by the lexical analyser.

Definition A syntax directed definition is an augmented context free grammar, where each production $A \to \alpha$, is associated with a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$, where f is a function, and either

1. b is a attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of α. In this case b is called a ***synthesized attribute.*** Or

2. b is an attribute of one of the grammar symbols α, and $c_1, c_2, \ldots, c_k$ are attributes belonging to A or α. In this case b is called an ***inherited attribute.***

In either case, we say that the attribute b ***depends*** on the attributes $c_1, c_2, \ldots, c_k$. In the previous example, $S.code$, $E.code$, and $E.place$ were all synthesized attributes, since they were all expressed solely in terms of attributes of grammar symbols on the right hand side of their productions.

In the context of compilattion, why do we need inherited attributes? Because there are static analyses which are naturally expressible in terms of inherited attributes. As a first example, consider a C-style declaration:

$$
\begin{aligned}
&var\_dec \to T : L \\
&L \to L_1, id \\
&L \to id \\
&T \to integer \\
&T \to real
\end{aligned}
$$

If we wanted to record the type of each variable, we could use the semantic rules shown below. Notice that in these rules $L.type$ and $id.type$ are inherited attributes.

$var\_dec \rightarrow T : L$	$\{\ L.type := T.type\}$
$L \rightarrow L_1, id$	$\{L_1.type := L.type,$
	$id.type := L.type\}$
$L \rightarrow id$	$\{id.type := L.type\}$
$T \rightarrow integer$	$\{T.type := integer\}$
$T \rightarrow real$	$\{T.type := real\}$

As a second example, consider a (simplified) procedure declaration

$$p\_decl \rightarrow p\_name\ var\_decl\ body$$

Suppose we wanted to find the list of variables visible (environment)inside the body of a procedure. This will be the list of variables visible just outside the procedure declaration, augmented by the variables declared locally in the procedure itself. In such a case, we can have the following attributes:

$p\_decl.env$ –	environment just outside the procedure declaration.
$body.env$ –	environment in the body.
$var\_decl.list$ –	list of local variables declared

Clearly, the required syntax directed definition is:

$$p\_decl \rightarrow p\_name\ var\_decl\ body \qquad \{body.env :=\ p\_decl.env \cup\ var\_decl.list\}$$

Again, $body.env$ is an inherited attribute.

2.2 Translation schemes

In the context of a syntax directed definition, a semantic rule specifies how different attributes are related. We can also look at a semantic rule as a prescription for computing an attribute. Following this point of view, we shall call semantic rules as ***actions***. If attribute evaluations are to take place along with parsing, a natural question to ask is the exact point of time during parsing when actions are executed. A ***translation scheme*** is a refinement of a syntax directed definition in which the exact point in time when the actions are to be executed is specified. This is done by embedding the actions within the right hand side of productions. In the translation

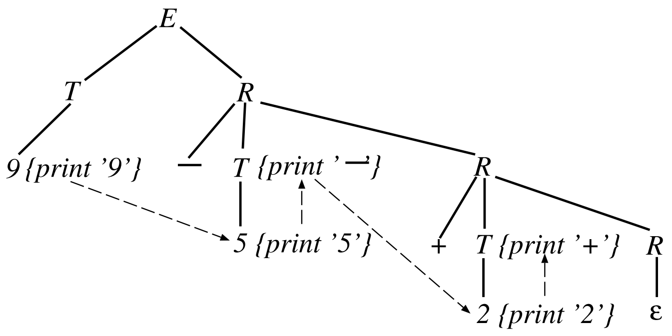


Figure 2: Execution of actions in a translation scheme.

scheme $A \to X_1 \ldots X_i, \{action\} X_{i+1} \ldots X_n$, *action* is executed after completion of the parse for X_i. and before the begining of X_{i+1}.

We now give an example which might not be directed related to compilation, but serves to illustrate that the choice of places where actions are to be embedded in a translation scheme is important. In the translation scheme shown below, the grammar generates expressions in infix form, whereas the actions converts them into their postfix forms.

$$E \to TR$$
$$R \to addop\ T\ \{print(addop.lexeme)\}\ R_1 \mid \epsilon$$
$$T \to num\ \{print(num.val)\}$$

The order of execution of actions for the string `9-5+2` is shown by dotted arrows in figure 2. By following the dotted arrows, we get the expression `95-2+`.

2.2.1 Restrictions on attribute dependences

Can an attribute of a grammar symbol in a production be defined in terms of attributes of any other grammar symbols of the same production? If we want to evaluate attributes along with parsing, thencertain restrictions must be placed on the dependences between them.

To see the reason for this observe that both top-down parsers or bottom-up parsers would construct the parse tree shown in the figure 3 in the order T_1, T_2, T_3, T_4. Clearly, some attribute dependences cannot be permitted. For instance,an attribute of T_2 cannot depend on any attribute of T_3. This is because the parse tree for T_3 will not have created during the evaluation of an attribute of T_2. The extent of restriction gives rise to two classes of translation schemes:

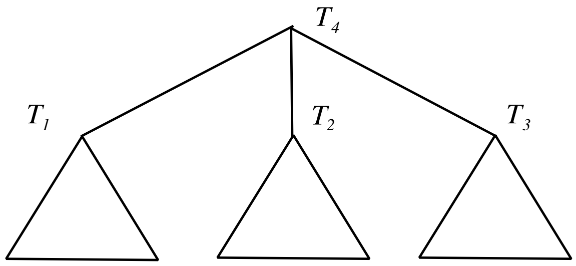


Figure 3: Order of parse tree creation

a. S-attributed scheme.

b. L-attributed scheme.

Definition A translation scheme is *S-attributed* if (i) every grammar symbol has synthesized attributes only, and (ii) all actions occur on the right hand side of productions.

The syntax directed definition shown below is an example of an S-attributed definition:

$$
\begin{array}{ll}
S \to id := E & S.code := E.code \parallel gen(id.place\ ':='\ E.place); \\
E \to E_1 + E_2 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\qquad ':='\ \ E_1.place\ '+'\ \ E_2.place); \\
E \to E_1 * E_2 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel E_2.code \parallel gen(E.place \\
 & \qquad\qquad ':='\ \ E_1.place\ '*'\ \ E_2.place); \\
E \to -E_1 & E.place := newtemp; \\
 & E.code\ := E_1.code \parallel gen(E.place\ ':='\ \ 'uminus' E_1.place); \\
E \to id & E.place := id.place; \\
 & E.code\ :=\ '\ '; \\
\end{array}
$$

Since S-attributed schemes do not permit inherited attributes, they are too restrictive. Therefore L-attributed schemes are used.

Definition: A translation scheme is L-attributed, if each inherited attribute of X_j, $1 \leq j \leq n$, on the right hand side of $A \to X_1X_2\ldots X_n$, depends only on (i) the attributes of the symbols $X_1, \ldots, X_{j-1}$, and (ii) the inherited attributes of A.

It follows from the definition that every S-attributed scheme is also a L-attributed scheme. The flow of attribute information in a L-attributed

scheme is shown in figure 4. An inherited attribute such as $T_2.inh$ gets its value from its left siblings and the inherited attribute of its parent. Whereas a synthesized attribute such as $T_4.syn$ gets its value solely from its children.

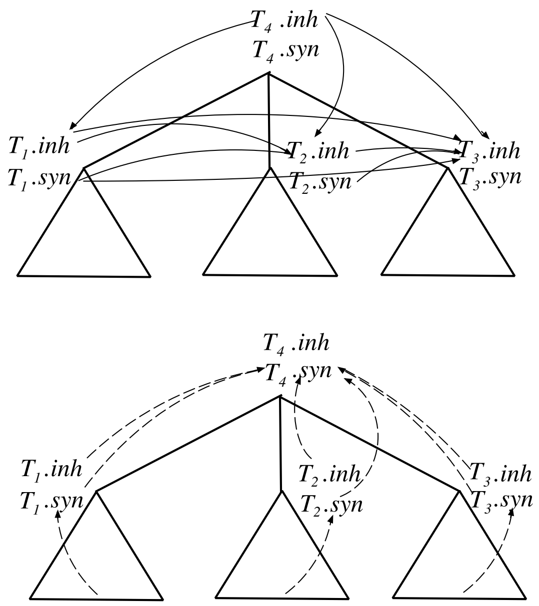


Figure 4: Flow of attributes in a L-attributed scheme

The dependences among the attributes (inherited and synthesized) of the nodes of a given parse tree is represented by a graph called the ***dependence graph.*** This graph has a node for each attribute of each node of the parse tree. In addition, there is an edge from the node for c to b, if the attribute b depends on c. The dependence graph for the syntax directed definition and the input string `a,b: real` is shown in figure 5. In the figure, the dotted lines represent edges of the parse tree, whereas the continuous lines denote dependence edges.

$var\_dec \rightarrow T : L$	$\{\ L.type := T.type\}$
$L \rightarrow L_1, id$	$\{L_1.type := L.type$
	$id.type := L.type\}$
$L \rightarrow id$	$\{id.type := L.type\}$
$T \rightarrow integer$	$\{T.type := integer\}$
$T \rightarrow real$	$\{T.type := real\}$

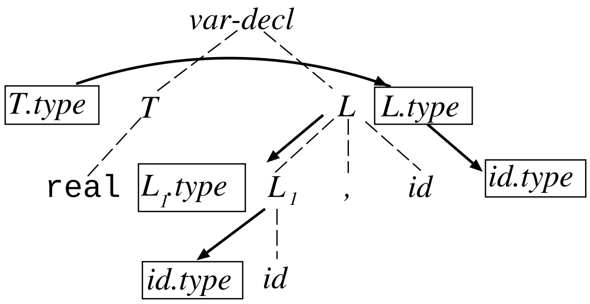


Figure 5: Dependence graph

3 Implementation of S-attributed scheme

We shall now discuss how to implement a S-attributed scheme with a bottom-up parsing strategy. To store attribute values, we augment the parsing stack with a second stack called value stack, represented as an array val. The correspondence between grammar symbols in the parsing stack and attributes in the value stack is that if location i in the parsing stack contains a grammar symbol A, then $val[i]$ will contain all the attributes of A. So each location in the value stack is assumed to be wide enough to contain all the attributes of any grammar symbol.

Figure 6 shows the parser and value stack configurations before and after a reduction. Prior to a reduction using a production $A \rightarrow XYZ$, the parser stack will contain grammar symbols X, Y and Z, and the value stack will contain the attributes $X.b$, $Y.c$ and $Z.d$. If an attribute a of A is defined as $f(X.b, Y.c, Z.d)$, then, after the reduction the parser stack will contain A and the value stack will contain $A.a$.

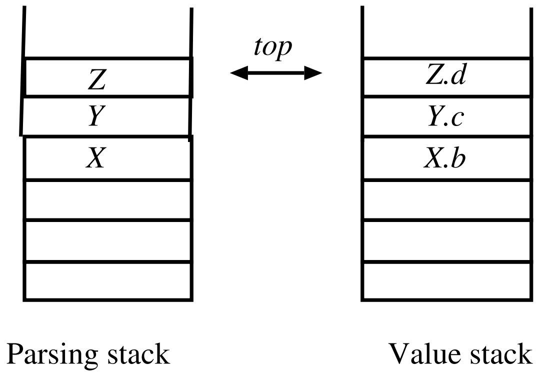


Figure 6: Implementation of a S-attributed scheme. (a) Before a reduction. (b) After a reduction

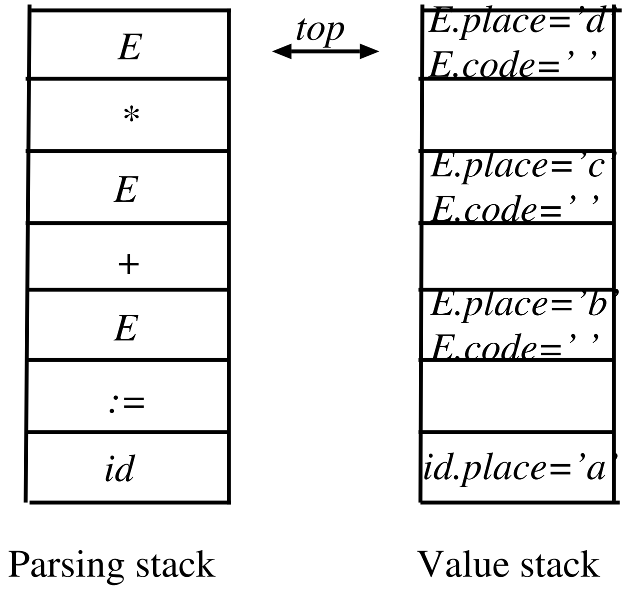
We can now replace the semantic rules by actions which refer to the actual data structures used to hold and manipulate the attributes.

Let each element of val **be a record with two fields** $val[i].code$ **and** $val[i].place$**. Then the syntax directed definition is;**

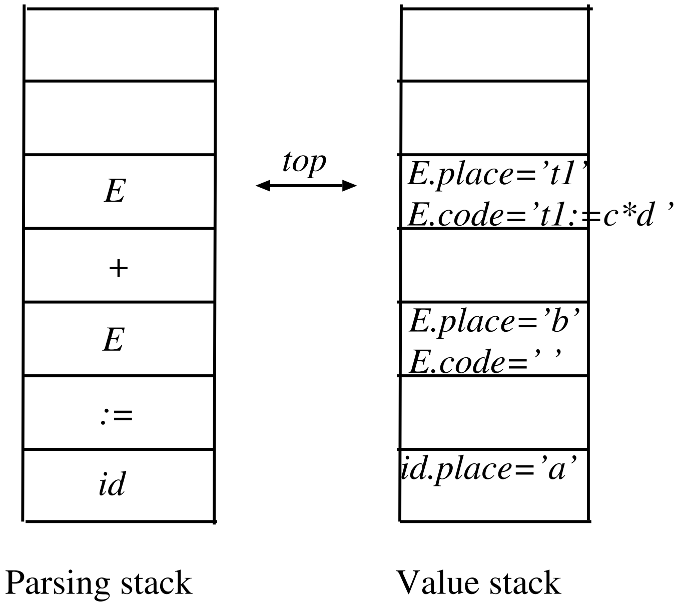
$$
\begin{array}{ll}
S \rightarrow id := E & val[top-2].code := val[top].code \parallel gen(val[top-2].place \\
& \qquad\qquad\quad ' :=' \ val[top].place); \\
E \rightarrow E_1 + E_2 & T := newtemp; \\
& val[top-2].code \ := \ val[top-2].code \parallel val[top].code \parallel \\
& \qquad\qquad\qquad gen(T \ ' :=' val[top-2].place \ '+' \\
& \qquad\qquad\qquad\qquad\quad val[top].place); \\
& val[top-2].place := T; \\
E \rightarrow E_1 * E_2 & T := newtemp; \\
& val[top-2].code \ := \ val[top-2].code \parallel val[top].code \parallel \\
& \qquad\qquad\qquad gen(T \ ' :=' val[top-2].place \ '*' \\
& \qquad\qquad\qquad\qquad\quad val[top].place); \\
& val[top-2].place := T; \\
E \rightarrow -E_1 & T := newtemp; \\
& val[top-1].code \ := \ val[top].code \parallel \\
& \qquad\qquad\qquad gen(T \ ' :=' \ \ 'uminus' \ val[top].place); \\
& val[top-2].place := T; \\
E \rightarrow id & val[top].place := val[top].place; \\
& val[top].code \ := \ ' \ '; \\
\end{array}
$$

S-ATTRIBUTED DEFINITIONS

EXAMPLE Before reduction using the production $E \to E_1 * E_2$.



After reduction using $E \to E_1 * E_2$.



S-ATTRIBUTED DEFINITIONS

Carefully distinguish between

1. ***Syntax directed definition*** **A context free grammar augmented with semantic rules. No assumptions are made regarding when the semantic rules are to be evaluated.**

2. ***Attribute grammar*** **A syntax directed definition in which the functions used in the semantic rules are free of side effects.**

3. ***Translation scheme*** **A syntax directed definition with actions instead of semantic rules. Actions are embedded within the right hand side of productions, their positions indicating the time of their evaluation during a parse.**

4. ***Implementation of a syntax directed definition*** **A translation scheme in which the actions are described in terms of actual data structures (e.g. value stack) instead of attribute symbols.**

L-ATTRIBUTED DEFINITIONS

Since S-attributed definitions do not permit inherited attributes, they are too restrictive. Therefore L-attributed definitions are used.

DEFINITION: A syntax directed definition is L-attributed, if each inherited attribute of X_j, $1 \leq j \leq n$, on the right hand side of $A \rightarrow X_1 X_2 \ldots X_n$, depends only on

1. **The attributes of the symbols $X_1, \ldots, X_{j-1}$.**
2. **The inherited attributes of A.**

The flow of attribute information in a L-attributed definition is shown below:

NOTE: Every S-attributed definition is also a L-attributed definition.

L-ATTRIBUTED DEFINITIONS

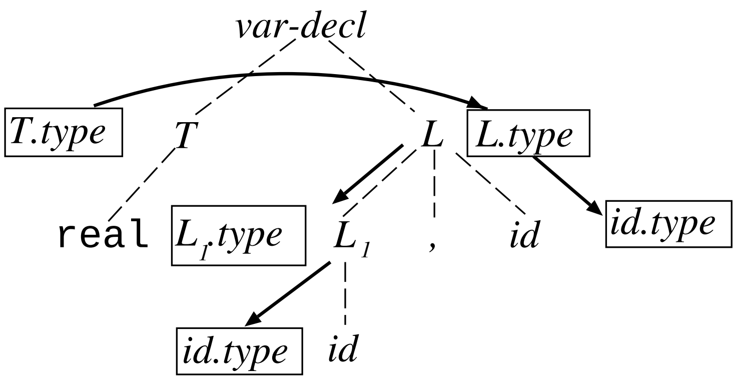
DEPENDENCE GRAPH

The dependences among the attributes (inherited and synthesized) of the nodes of a given parse tree is represented by a graph called the *dependence graph.*

This graph has a node for each attribute of each node of the parse tree. In addition, there is an edge from the node for c to b, if the attribute b depends on c.

The dependency graph for the syntax directed definition and the input string `a,b:  real` **is shown below:**

$var\_dec \rightarrow T : L$	$\{\ L.type := T.type\}$
$L \rightarrow L_1, id$	$\{L_1.type := L.type$
	$id.type := L.type\}$
$L \rightarrow id$	$\{id.type := L.type\}$
$T \rightarrow integer$	$\{T.type := integer\}$
$T \rightarrow real$	$\{T.type := real\}$



In the diagram above, the dotted lines represent edges of the parse tree, whereas the continuous lines denote dependence edges.

L-ATTRIBUTED DEFINITIONS

RECURSIVE DESCENT WITH L-ATTR. EVALUATION

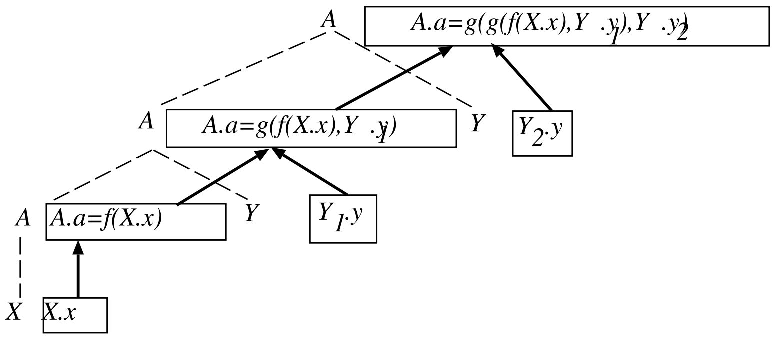
If we have a translation scheme whose underlying grammar involves left recursion, what happens to the actions when we transform the grammar to remove left recursion?

There is a procedure to transform the actions. But the procedure works provided the original translation scheme has synthesized attributes only.

A generic translation scheme involving a grammar with left recursion:

$$\begin{array}{ll} A \rightarrow A_1 Y & \{\ A.a := g(A_1.a, Y.y)\} \\ A \rightarrow X & \{A.a := f(X.x)\} \end{array}$$

The dependency graph for the string XYY would be:



L-ATTRIBUTED DEFINITIONS

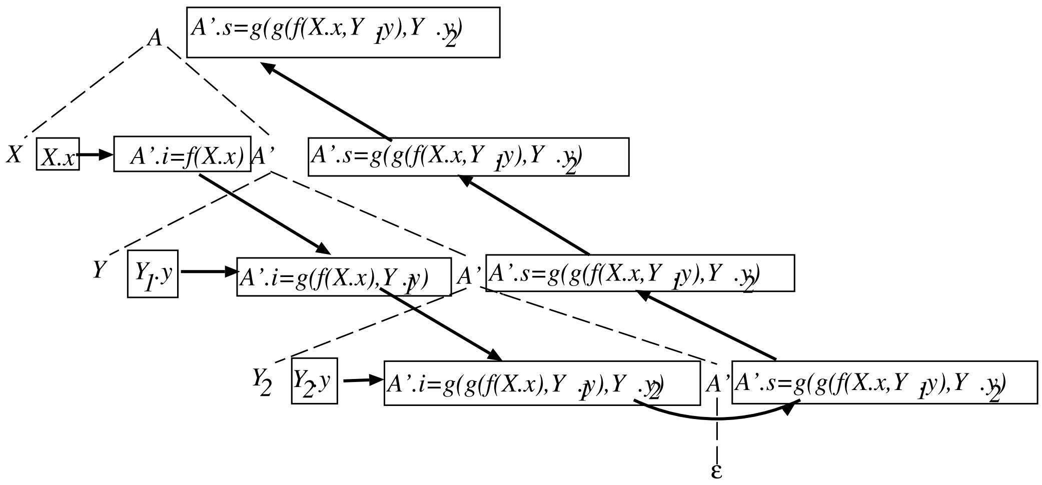
After removing left recursion, the grammar becomes:

$$A \to XA'$$
$$A' \to YA' \mid \epsilon$$

Transform the actions in the manner shown:

$$A \to X\ \{A'.i := f(X.x)\}\ A'\ \{A.a := A'.s\}$$
$$A' \to Y\ \{A'_1.i := g(A'.i, Y.y)\}\ A'_1\ \{A'.s := A'_1.s\}$$
$$A' \to \epsilon\ \{A'.s := A'.i\}$$

The new dependency graph, which calculates the same attribute at the root is:



L-ATTRIBUTED DEFINITIONS

EXAMPLE

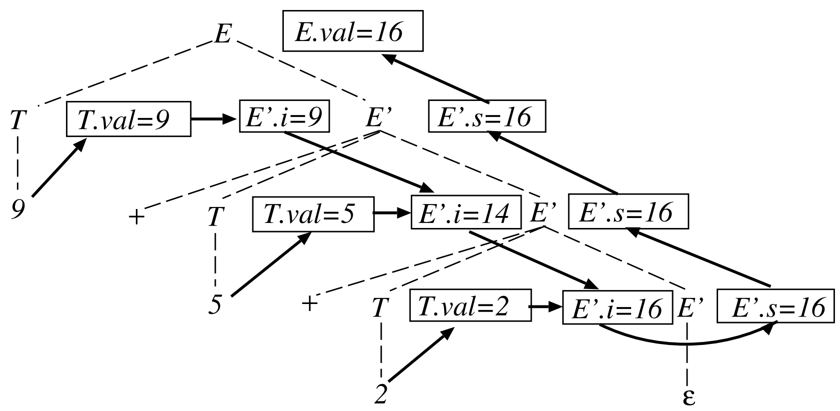
The translation scheme

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \{\ E.val := E_1.val + T.val\} \\
E \rightarrow T & \{E.val := T.val\} \\
T \rightarrow num & \{T.val := num.val\}
\end{array}
$$

is transformed into

$$
\begin{array}{l}
E \rightarrow T\ \{E'.i := T.val\}\ E'\ \{E.val := E'.s\} \\
E' \rightarrow +\ T\ \{E'_1.i := E'.i + T.val\}\ E'_1\ \{E'.s := E'_1.s\} \\
E' \rightarrow \epsilon\ \{E'.s := E'.i\} \\
T \rightarrow num\ \{T.val := num.val\}
\end{array}
$$

For the input string $9 + 5 + 2$, the dependency graph will be



L-ATTRIBUTED DEFINITIONS

RECURSIVE DESCENT WITH L-ATTR EVALUATION

ALGORITHM This algorithm takes as input a translation scheme, whose underlying grammar is suitable for predictive parsing, and produces code for a predictive parser.

1. For each nonterminal A, construct a function that has

   a. A formal parameter for each inherited attribute of A.

   b. Returns the synthesized attributes of A as a record.

   c. Has a local variable for each attribute of each grammar symbol occurring in any production of A.

2. The body of the function will depend on the productions with A on the left hand side. Examine the production $A \to X_1 \ldots X_n$. For each X_i do the following:

   a. If X_i is a token with synthesized attribute x, save the value of x in the variable for $X_i.x$. Then generate a call to match X_i and advance the input pointer.

   b. If X_i is a non-terminal, generate an assignment statement $c := X_i(b_1, \ldots, b_k)$, where $b_1, \ldots, b_k$ are the variables for the inherited attributes of X_i and c is the variable for the synthesized attribute of X_i.

   c. For an action, copy the code into the parser, replacing each reference to an attribute by the variable for the attribute.

L-ATTRIBUTED DEFINITIONS

EXAMPLE

The code for

$$
\begin{array}{l}
E' \rightarrow +T\ \{E'_1.i := E'.i + T.val\}\ E'_1\ \{E'.s := E'_1.s\} \\
E' \rightarrow \epsilon\ \{E'.s := E'.i\}
\end{array}
$$

is

```
function E_PRIME(e_prime_i: integer): integer;
   var t_val, e1_prime_i, e1_prime_s: integer;
   begin
       if nextsymbol = '+' then
         begin
             t_val := T;
             e1_prime_i := eprime_i + t_val;
             e1_prime_s := E1_PRIME(e1_prime_i);
             E_PRIME := e1_prime_s
         end
         else E_PRIME := e_prime_i
   end;
```

L-ATTRIBUTED DEFINITIONS

BOTTOM-UP PARSING WITH L-ATTR. EVALUATION

Given a translation scheme, we shall transform it so that all actions appear to the right of productions. This is because we want the attribute evaluation to take place *just before a reduction.*
This is done by the use of marker non-terminals. For example

$$
\begin{array}{l}
E \rightarrow TR \\
R \rightarrow addop\ T\ \{print(addop.lexeme)\}\ R_1 \mid \epsilon \\
T \rightarrow num\ \{print(num.val)\}
\end{array}
$$

is changed into

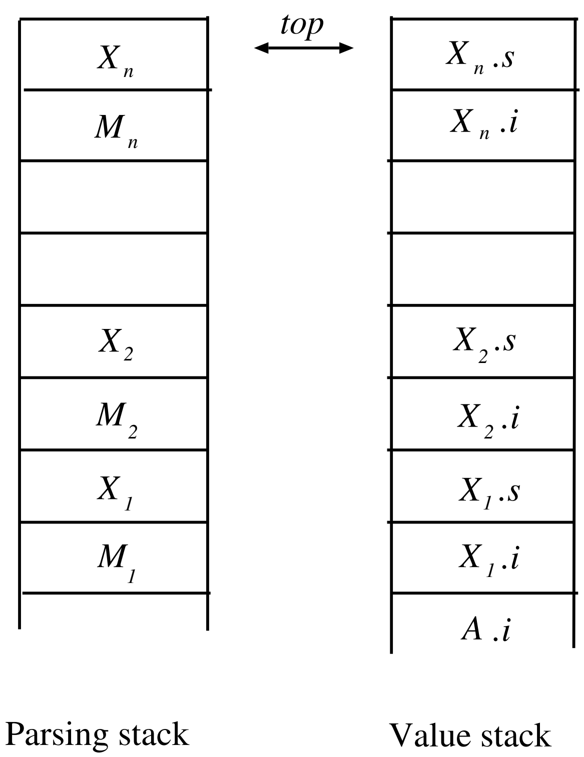
$$
\begin{array}{l}
E \rightarrow TR \\
R \rightarrow addop\ T\ M\ R_1 \mid \epsilon \\
T \rightarrow num\ \{print(num.val)\} \\
M \rightarrow \epsilon\ \{print(addop.lexeme)\}
\end{array}
$$

Note that there is a reference to an attribute of a grammar symbol $addop.lexeme$ **in the production** $M \rightarrow \epsilon$**, which does not have an occurrence of** $addop$**. This is different from the examples considered earlier.**
However, the method of implementation ensures that the positions of such attributes, which are non-local to a production, can be determined in the value stack.

L-ATTRIBUTED DEFINITIONS

The key idea is that a production $A \rightarrow X_1, \ldots, X_n$ is transformed into $A \rightarrow M_1X_1, \ldots, M_nX_n$ by the introduction of marker non-terminals M_i.

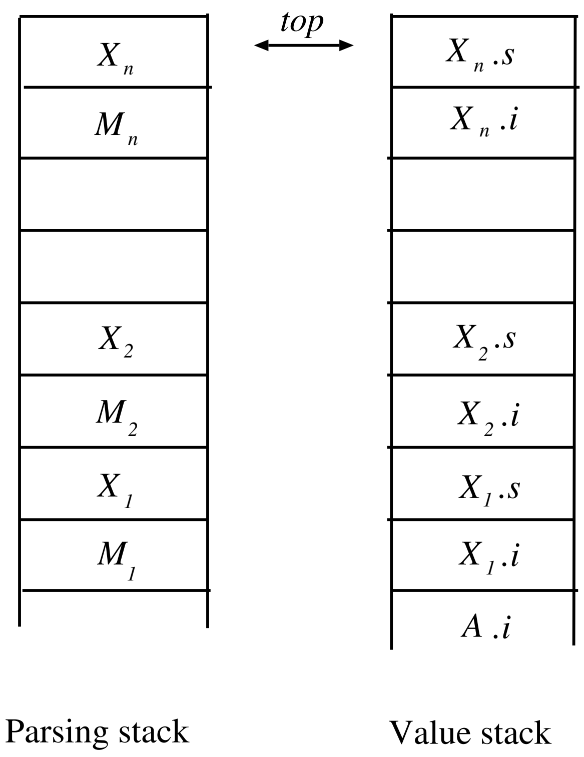
1. **The synthesized attribute $X_j.s$ will go on the value stack (*val array*) in the position associated with the entry X_j.**
2. **The inherited attribute $X_j.i$, if there is one, appears in the same array, associated with the marker non-terminal M_j.**



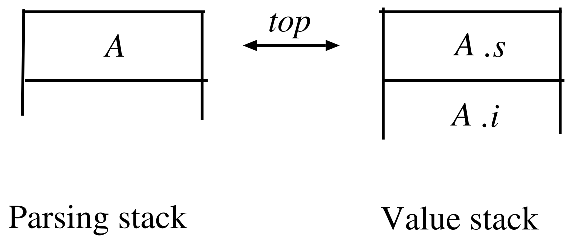
Where do we find $A.i$? It is found in the position immediately below M_1 in the value stack. This is always true for the inherited attribute of the nonterminal of the left hand side of the production.

L-ATTRIBUTED DEFINITIONS

Just before a reduction using $A \rightarrow M_1X_1, \ldots, M_nX_n$, the parsing stack and the value stack will be



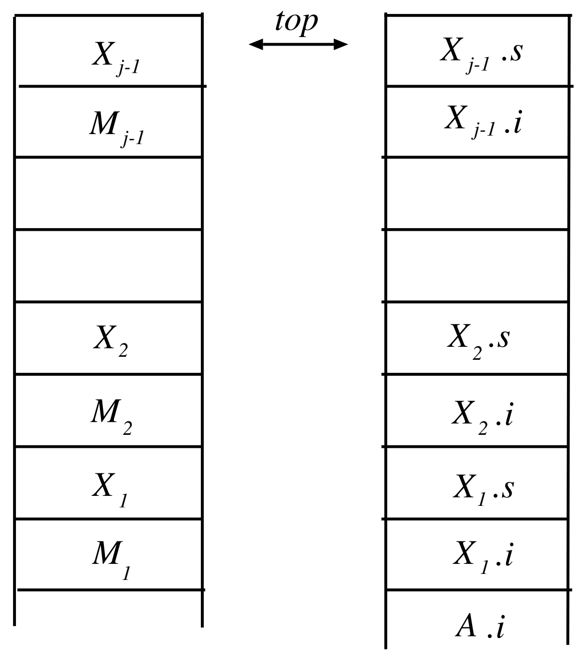
After the reduction the configuration is



If we are to evaluate the attributes just prior to a reduction, can these attributes be located in the value stack?

L-ATTRIBUTED DEFINITIONS

1. **Consider a reduction using $M_j \to \epsilon$ along with an action to be performed to evaluate an inherited attribute of X_j. The stack configurations are**



Parsing stack Value stack

If the action corresponding to $M_j \to \epsilon$ refers to

a. $A.i$, **it would be found in the location** $top - 2j - 2$.

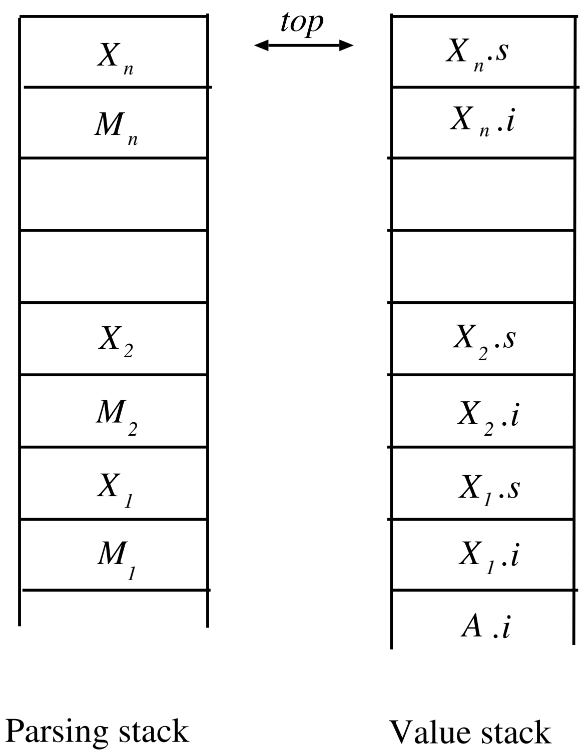
b. $X_1.i$, **it would be found in the location** $top - 2j - 1$.

c. $X_1.s$, **it would be found in the location** $top - 2j$.

So the reduction can be done, and the value of $X_j.i$ put in $top + 1$.

L-ATTRIBUTED DEFINITIONS

2. Now consider a reduction using $A \to M_1X_1, \ldots, M_nX_n$ along with an action which computes a synthesized attribute of A. The stack configurations in this case are



If the action corresponding to $M_j \to M_1X_1, \ldots, M_nX_n$ refers to

a. $A.i$, it would be found in location $top - 2n - 2$.

b. $X_1.i$, it would be found in location $top - 2n - 1$.

Once again the reduction can be done, and the value of $A.s$ put in $top - 2n - 1$.

L-ATTRIBUTED DEFINITIONS

Consider the example shown below

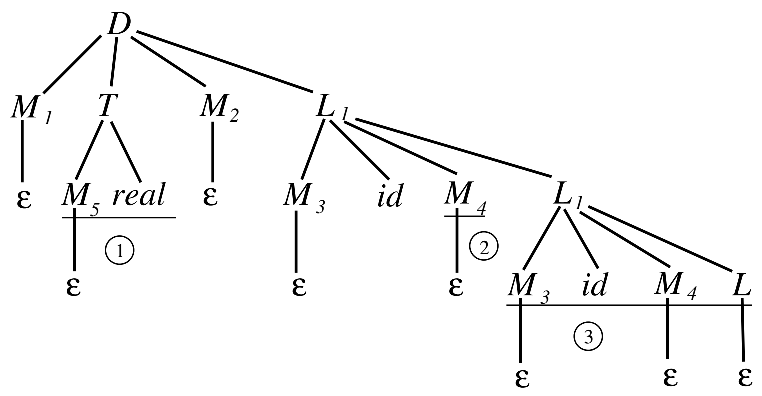
$$
\begin{array}{ll}
D \rightarrow T & \{\ L.in := T.type\} \\
\qquad L & \\
L \rightarrow & \{id.type := L.in\} \\
\qquad id & \{L_1.in := L.in\} \\
\qquad L_1 & \\
L \rightarrow \epsilon & \\
T \rightarrow real & \{T.type := real\}
\end{array}
$$

The grammar will be transformed into

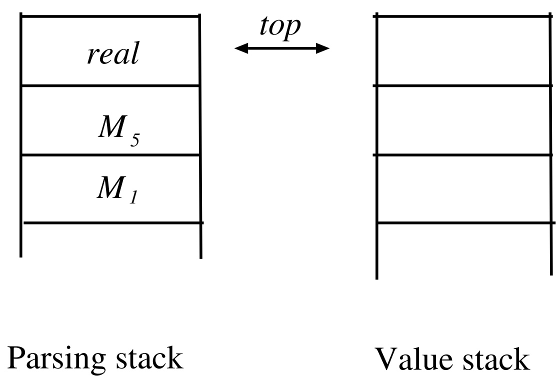
$$
\begin{array}{ll}
D \rightarrow M_1\ T\ M_2\ L & \\
L \rightarrow M_3\ id\ M_4\ L_1 & \\
L \rightarrow \epsilon & \\
T \rightarrow M_5\ real & \{T.type := real\} \\
M_1 \rightarrow \epsilon & \\
M_2 \rightarrow \epsilon & \{L.in := T.type\} \\
M_3 \rightarrow \epsilon & \{id.type := L.in\} \\
M_4 \rightarrow \epsilon & \{L_1.in := L.in\} \\
M_5 \rightarrow \epsilon &
\end{array}
$$

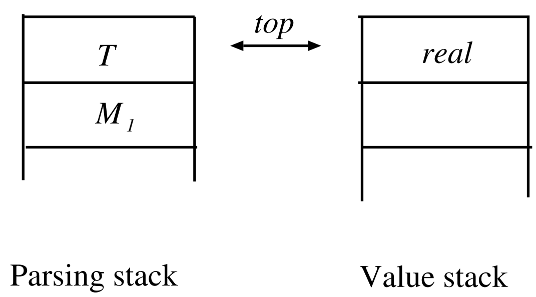
L-ATTRIBUTED DEFINITIONS

The parse tree for the string `real a b` is

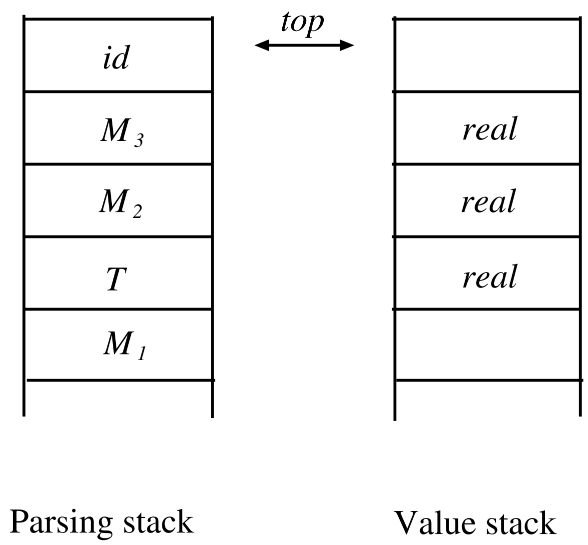


The parse and value stack configurations just before and after the reductions of the handles 1, 2 and 3, indicated in the figure are:
handle 1

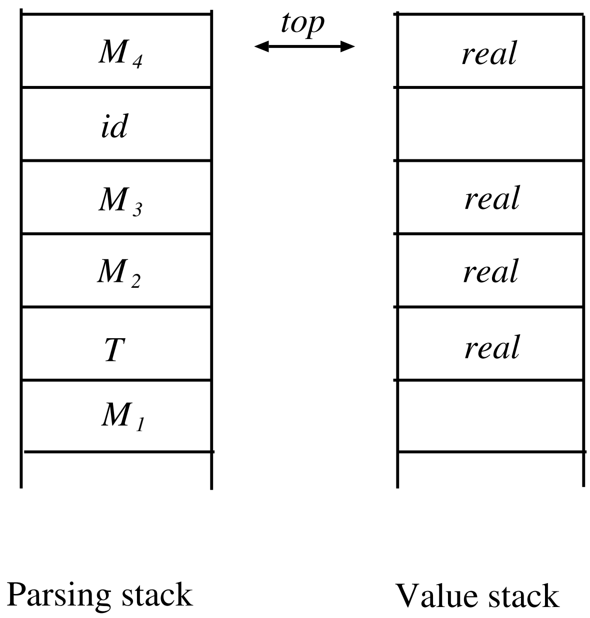




handle 2



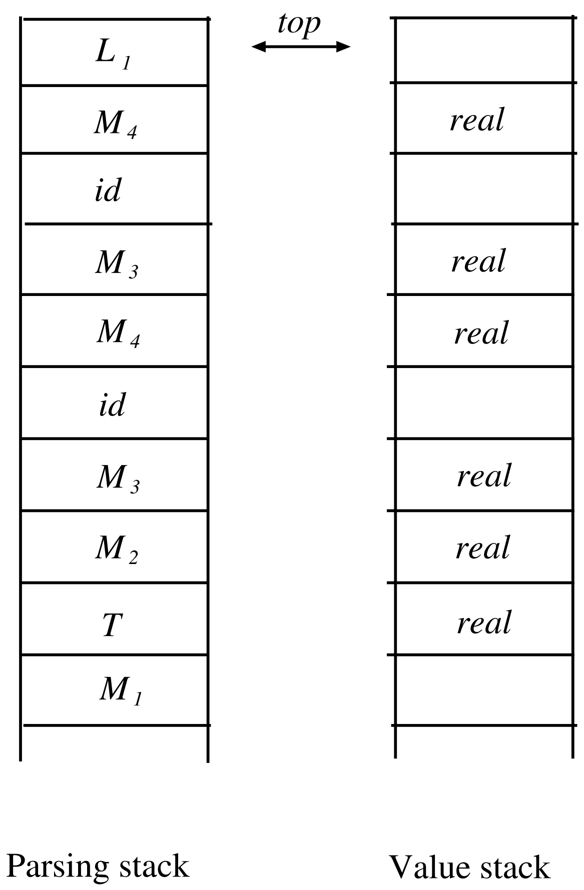
id
M 3
M 2
T
M 1
top
real
real
real
Parsing stack
Value stack



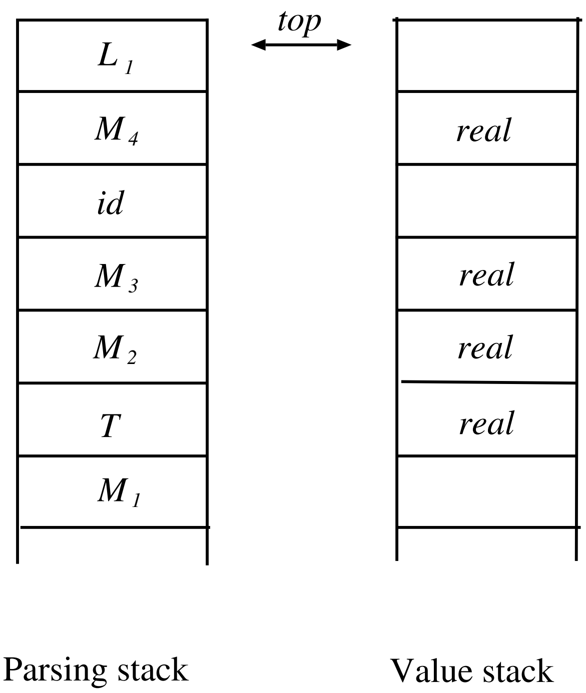
M 4
id
M 3
M 2
T
M 1
top
real
real
real
real
Parsing stack
Value stack

L-ATTRIBUTED DEFINITIONS

handle 3



L-ATTRIBUTED DEFINITIONS



4 References

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.

2. Bauer, F.L. and J. Eickel (1974): *Compiler Construction : An Advanced Course*, Springer-Verlag, Berlin.

3. Brown, P.J. (1979): *Writing Interactive Compilers and Interpreters*, Wiley, New York.

4. Fischer, C.N. and R.J. LeBlanc (1988) : *Crafting a compiler*, Benjamin/Cummings, Menlo Park, California.

5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.

6. Holub, A.I. (1993): *Compiler Design in C* Prentice-Hall, Englewood-Cliffs.

7. Levine, J.R., T. Mason and D. Brown (1990): *Lex and Yacc*, 2^{nd} edition, O'Reilly & Associates, Sebastopol.

8. Schreiner, A.T., H.G. Friedman Jr. (1985): *Introduction to Compiler Construction with UNIX* Prentice-Hall, Englewood Cliffs.

9. Tremblay, J.P. and P.G. Sorenson (1985): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

10. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.

11. Waite, W.M. and Goos, G. (1984) *Compiler Construction*, Springer-Verlag, New York.