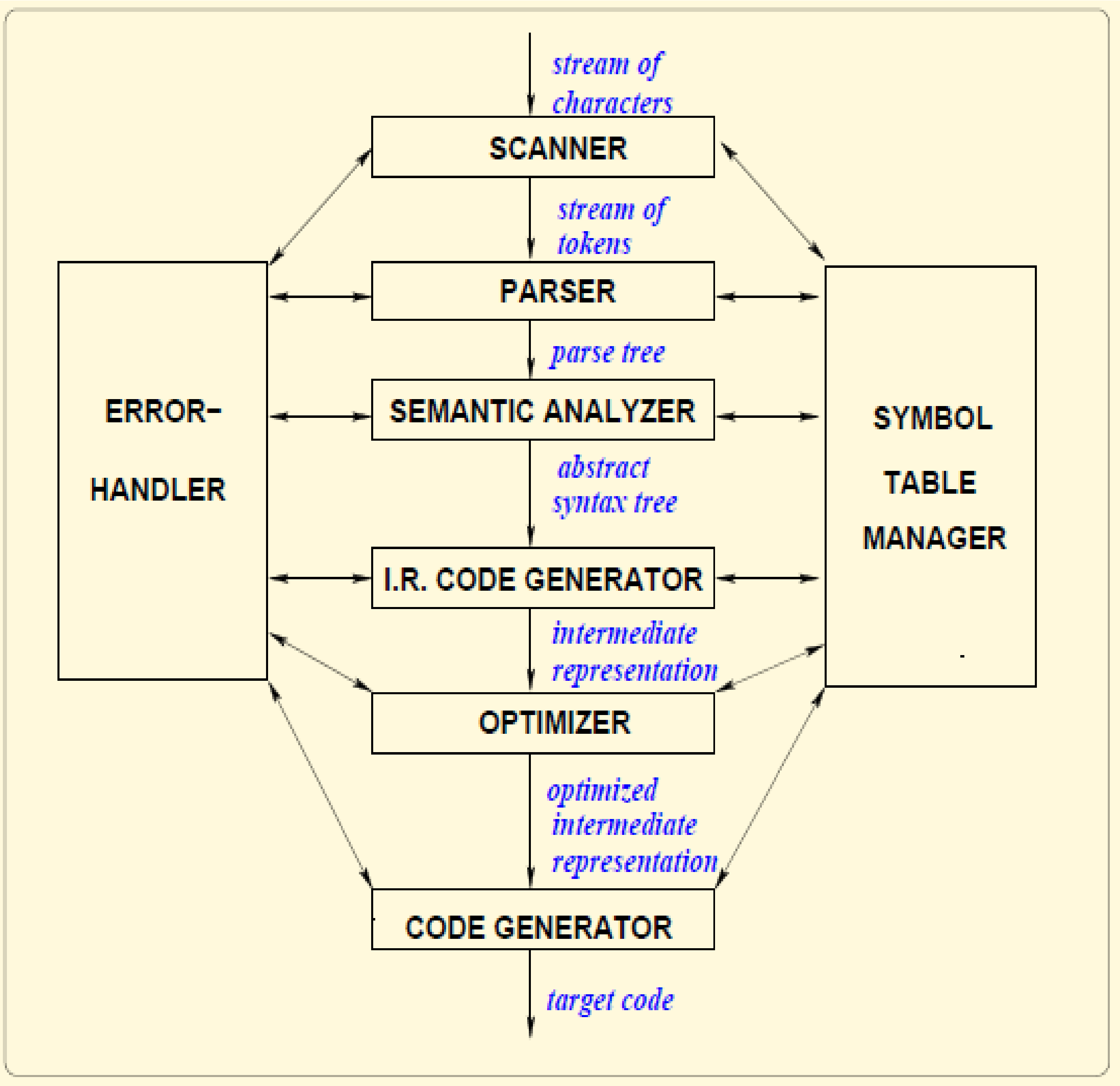
Semantic Analysis


stream of
characters
SCANNER
stream of
tokens
PARSER
parse tree
SEMANTIC ANALYZER
abstract
syntax tree
I.R. CODE GENERATOR
intermediate
representation
OPTIMIZER
optimized
intermediate
representation
CODE GENERATOR
target code
ERROR-
HANDLER
SYMBOL
TABLE
MANAGER

Semantic Analysis

- There are context-sensitive aspects of a program that cannot be represented/enforced by a context-free grammar definition.
- For example
  - correspondence between formal and actual parameters
  - type consistency between declaration and use.
  - scope and visibility issues with respect to identifiers in a program.

Semantic Analysis

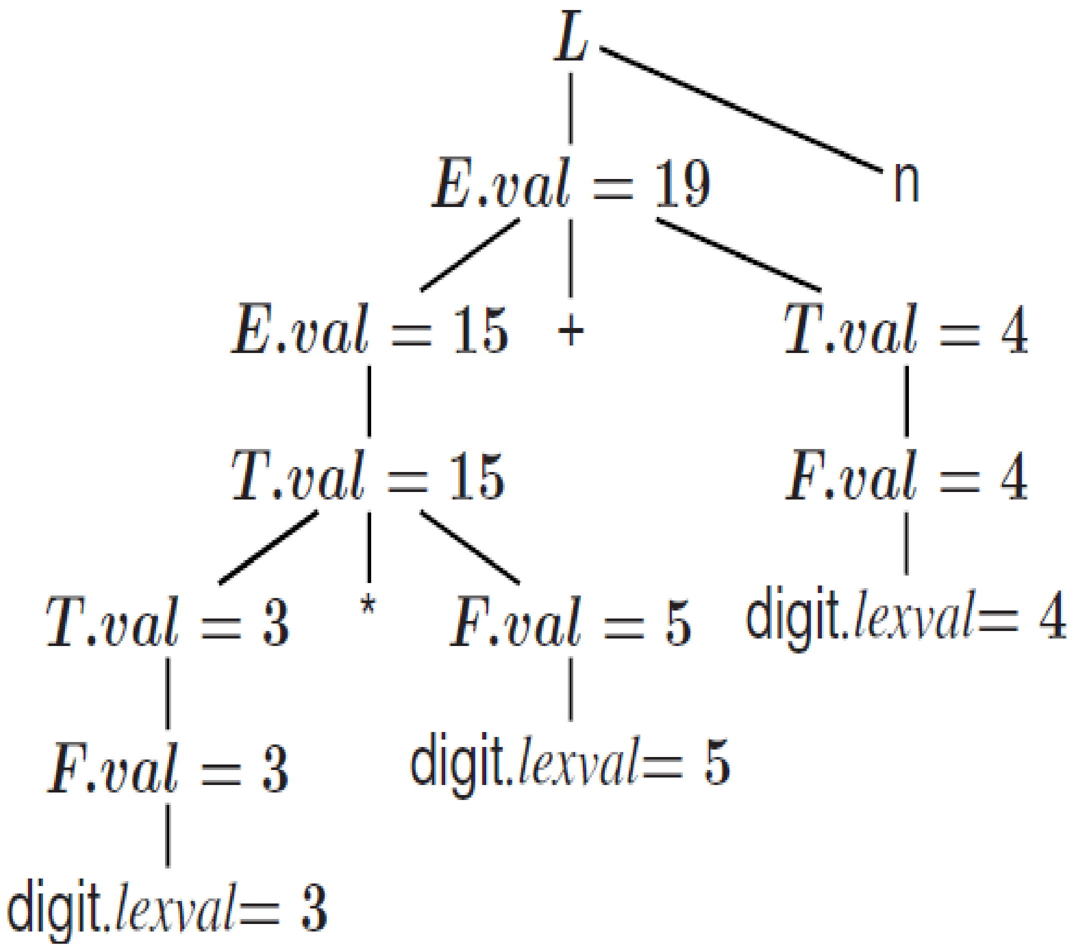
- To capture the context sensitive aspect, we need to understand
  - How to represent it (representation formalism)
  - How to implement it (implementation mechanism)
- As representation formalism we use *Syntax Directed Translations*
- Syntax Directed Translation relates an input sentence to its syntactic structure, i.e., to its Parse-Tree.
- We associate Attributes to the grammar symbols representing the language constructs.
- Values for attributes are computed by Semantic Rules associated with grammar productions.

Semantic Analysis

- Evaluation of Semantic Rules is used to:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages; etc.
- Two notations :
  - Syntax Directed Definitions. High-level specification hiding many implementation details (also called Attribute Grammars).
  - Translation Schemes. More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

- Grammar symbols associated with **attributes**
- Productions are associated with **Semantic Rules**
- Generates **Annotated Parse-Trees** where each node is a record with a field for each attribute



Syntax Directed Definitions

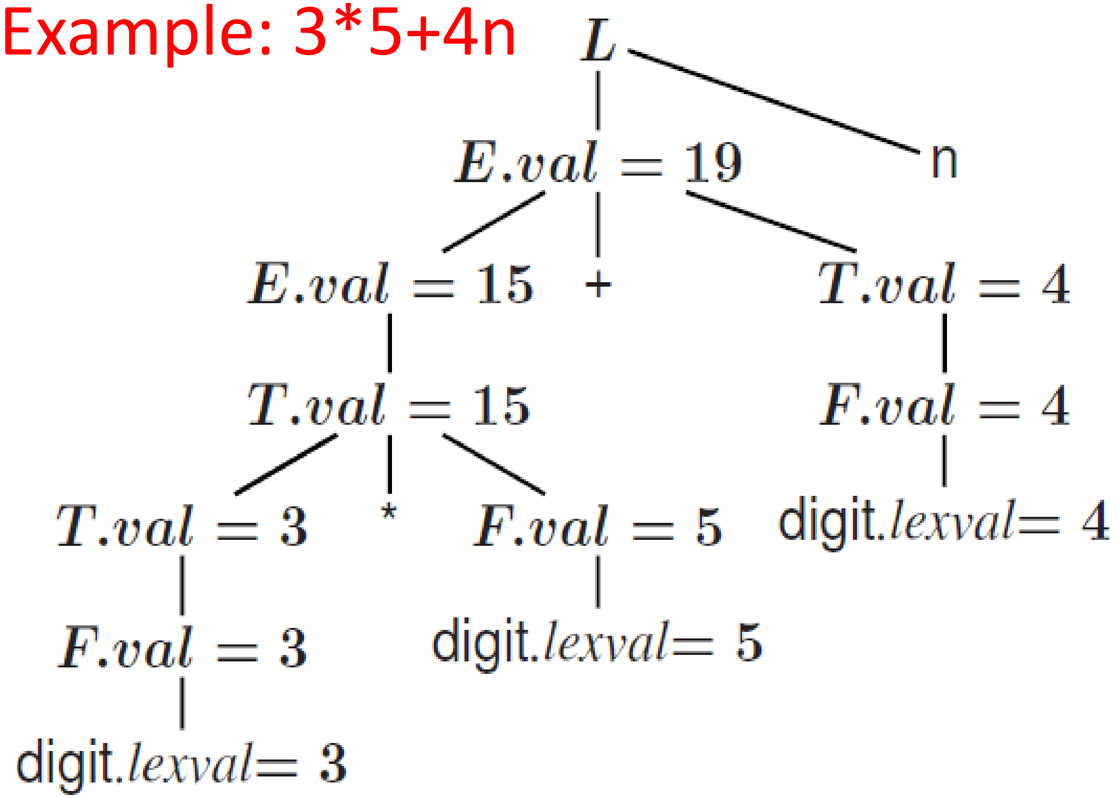
- Attributes are of two types:
- **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
- **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes.

S-Attributed Definitions

An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val :=\mathrm{digit}.lexval$

Example: 3*5+4n



L-Attributed Definitions

- A grammar is L-attributed if each attribute a_j at X_i of a grammar rule:

  X_0 -> X_1 X_2 … X_n

1) is either a synthesized attribute, or
2) the value of a_j at X_i only depends on attribute of the symbols

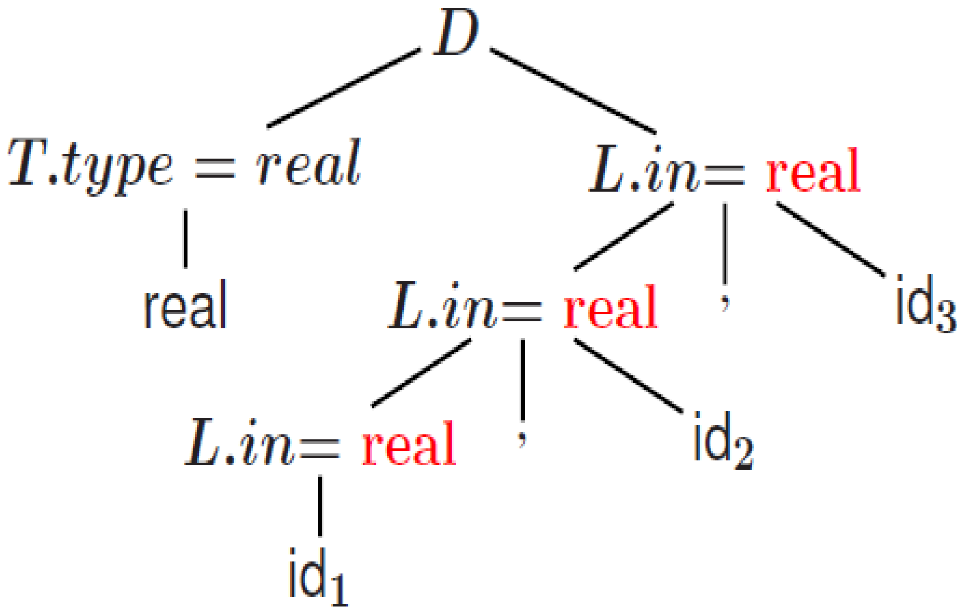
X_0, … X_{i-1}, that occur to the left of X_i in the grammar rule, or

3) depends on the *inherited* attributes of X_0.

L-Attributed Definitions

Example: real id1, id2, id3

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type :=$ *integer*
$T \rightarrow$ real	$T.type :=$ *real*
$L \rightarrow L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry*, *L.in*)
$L \rightarrow$ id	*addtype*(id.*entry*, *L.in*)



Implementing Attribute Evaluation

- **Evaluation of S-Attributed Definitions**
- **Using a bottom-up parser**
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction A -> α is made, the attribute for A is computed from the attributes of α which appear on the stack.

Extending a parser stack

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
$\cdots$	$\cdots$

PRODUCTION	CODE
$L \to E\mathrm{n}$	$print(val[top-1])$
$E \to E_1 + T$	$val[ntop] := val[top] + val[top-2]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top] * val[top-2]$
$T \to F$	
$F \to (E)$	$val[ntop] := val[top-1]$
$F \to \mathrm{digit}$	

INPUT	*state*	*val*	PRODUCTION USED
3*5+4**n**	–	–	
*5+4**n**	3	3	
*5+4**n**	F	3	$F \to$ **digit**
*5+4**n**	T	3	$T \to F$
5+4**n**	T *	3 _	
+4**n**	T * 5	3 _ 5	
+4**n**	T * F	3 _ 5	$F \to$ **digit**
+4**n**	T	15	$T \to T * F$
+4**n**	E	15	$E \to T$
4**n**	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \to$ **digit**
n	E + T	15 _ 4	$T \to F$
n	E	19	$E \to E + T$
	E **n**	19 _	
	L	19	$L \to E$ **n**

Implementing Attribute Evaluation

- **Evaluation of L-Attributed Definitions**
- Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree.
- L-Attributed Definitions can be evaluated by mixing PostOrder traversal for synthesized attributes and PreOrder traversal for inherited attributes.

Implementation using Translation Scheme

- **Definition.** A Translation Scheme is a context-free grammar in which

1. Attributes are associated with grammar symbols;
2. Semantic Actions are enclosed between braces {} and **are inserted within the right-hand side of productions.**
3. Yacc uses Translation Schemes.

Implementation using Translation Scheme

<u>Rules</u>

For S-attributed definitions – Put all semantic rules into braces at the right end of each production.

For L-attributed definitions –

1. An inherited attribute for a symbol on the right hand side of a production must be computed in an action before the symbol
2. An action must not refer to a synthesized attribute of a symbol that is to the right
3. A synthesized attribute for the non-terminals on the left hand side can only be computed after all attributes it references are already computed

Implementation using Translation Scheme

Examples

- Only synthesized attributes

  S→A1 A2 {S.s = A1.s + A2.s}

  A → a { A.s = 1 }
- Synthesized and inherited attributes

  S → {A1.in = 1; A2.in = 2} A1 A2

  A → a {A.s = 1}

OR

  S → {A1.in = 1} A1 {A2.in = 2} A2

  A → a {A.s = 1}

Grammar:

S → A A

A → a

Implementation using Translation Scheme

L	$\rightarrow$	$E\ \mathbf{n}$	$\{\ \text{print}(E.val);\ \}$
E	$\rightarrow$	$E_1 + T$	$\{\ E.val = E_1.val + T.val;\ \}$
E	$\rightarrow$	T	$\{\ E.val = T.val;\ \}$
T	$\rightarrow$	$T_1 * F$	$\{\ T.val = T_1.val \times F.val;\ \}$
T	$\rightarrow$	F	$\{\ T.val = F.val;\ \}$
F	$\rightarrow$	$(\ E\ )$	$\{\ F.val = E.val;\ \}$
F	$\rightarrow$	$\mathbf{digit}$	$\{\ F.val = \mathbf{digit}.lexval;\ \}$

Figure 5.18: Postfix SDT implementing the desk calculator

Implementation using Translation Scheme

$$D \rightarrow \quad T \ \{L.in := T.type\} \ L$$

$$T \rightarrow \quad \text{int} \ \{T.type := integer\}$$

$$T \rightarrow \quad \text{real} \ \{T.type := real\}$$

$$L \rightarrow \quad \{L_1.in := L.in\} \ L_1, \text{id} \ \{addtype(\text{id}.entry, L.in)\}$$

$$L \rightarrow \quad \text{id} \ \{addtype(\text{id}.entry, L.in)\}$$

Code generation for a “while”-statement using Translation Scheme

Grammar: S → while (C) S

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\,(\,C\,)\,S_1 & L1 = new(); \\
 & L2 = new(); \\
 & S_1.next = L1; \\
 & C.false = S.next; \\
 & C.true = L2; \\
 & S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

Figure 5.27: SDD for while-statements

S.next : beginning of the next code

S.Code : code for S ends with a jump to S.next

C.true : beginning of the code to be executed if C is true

C.false : beginning of the code to be executed if C is false

C.code : code for C with a jump to either C.true or C.false

Implementation using Translation Scheme

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (\ C\ )\ S_1 & L1 = new(); \\
 & L2 = new(); \\
 & S_1.next = L1; \\
 & C.false = S.next; \\
 & C.true = L2; \\
 & S.code = \textbf{label} \parallel L1 \parallel C.code
\end{array}
$$

Figure 5.27: SDD for while-statements

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\
\quad C\ ) & \{\ S_1.next = L1;\ \} \\
\quad S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}
\end{array}
$$

Figure 5.28: SDT for while-statements

L1 labels the beginning of while statement
and L2 labels the beginning of S1

Compile-Time Evaluation of Translation Schemes

Main Idea. Starting from a Translation Scheme (with embedded actions)

- introduce a transformation that makes all the actions occur at the right ends of their productions.
- For each embedded semantic action introduce a new Marker (i.e., a non terminal, say M) with an empty production ($M \rightarrow \varepsilon$).
- The semantic action is attached at the end of the production ($M \rightarrow \varepsilon$).

Compile-Time Evaluation of Translation Schemes

Production	Semantic Rules
S→ aAC	C.in = A.s
S→ bABC	C.in = A.s
C→ c	C.s = g(C.i)
………..	…..

Production	Semantic Rules
S→ aAM1C	M1.in = A.s C.in = M1.s
S→ bABM2C	M2.in = A.s C.in = M2.s
C→ c	C.s = g(C.i)
M1→ ε	M1.s = M1.in
M2→ ε	M2.s = M2.in
………	……..

Compile-Time Evaluation of Translation Schemes

- General rules to compute translations schemes during bottom-up parsing for an L-attributed grammar.
- For every production A → X1 . . .Xn introduce n new markers M1, . . . ,Mn and replace the production by A → M1X1 . . .MnXn.
- Thus, the position of every synthesized and inherited attribute of Xj and A are known:
  - Xj.s is stored in the *val* entry in the parser stack associated with Xj ;
  - Xj.i is stored in the *val* entry in the parser stack associated with Mj ;
  - A.i is stored in the *val* entry in the parser stack immediately before the position storingM1

Compile-Time Evaluation of Translation Schemes

- Computing the inherited attribute Xj.i after reducing Mj → ε

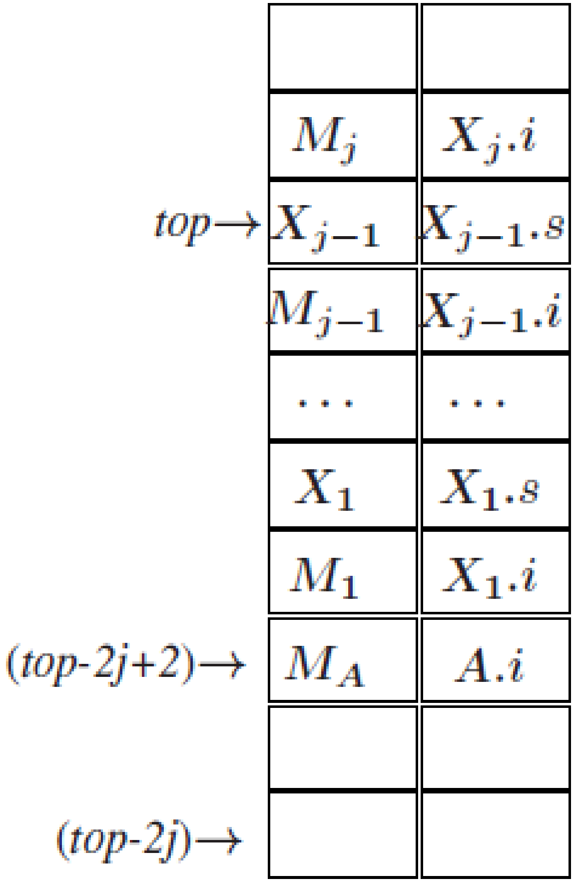
A.i is in val[top – 2j + 2]

X1.i is in val[top – 2j + 3]

X1.s is in val[top – 2j + 4]

X2.i is in val[top – 2j + 5]

And so on



Limitations of SDT

- Checking whether a variable is defined before its usage
- Checking the type and storage address of a variable
- Checking whether a variable is used or not

Need to use a symbol table : global data to show side effects of semantic actions.