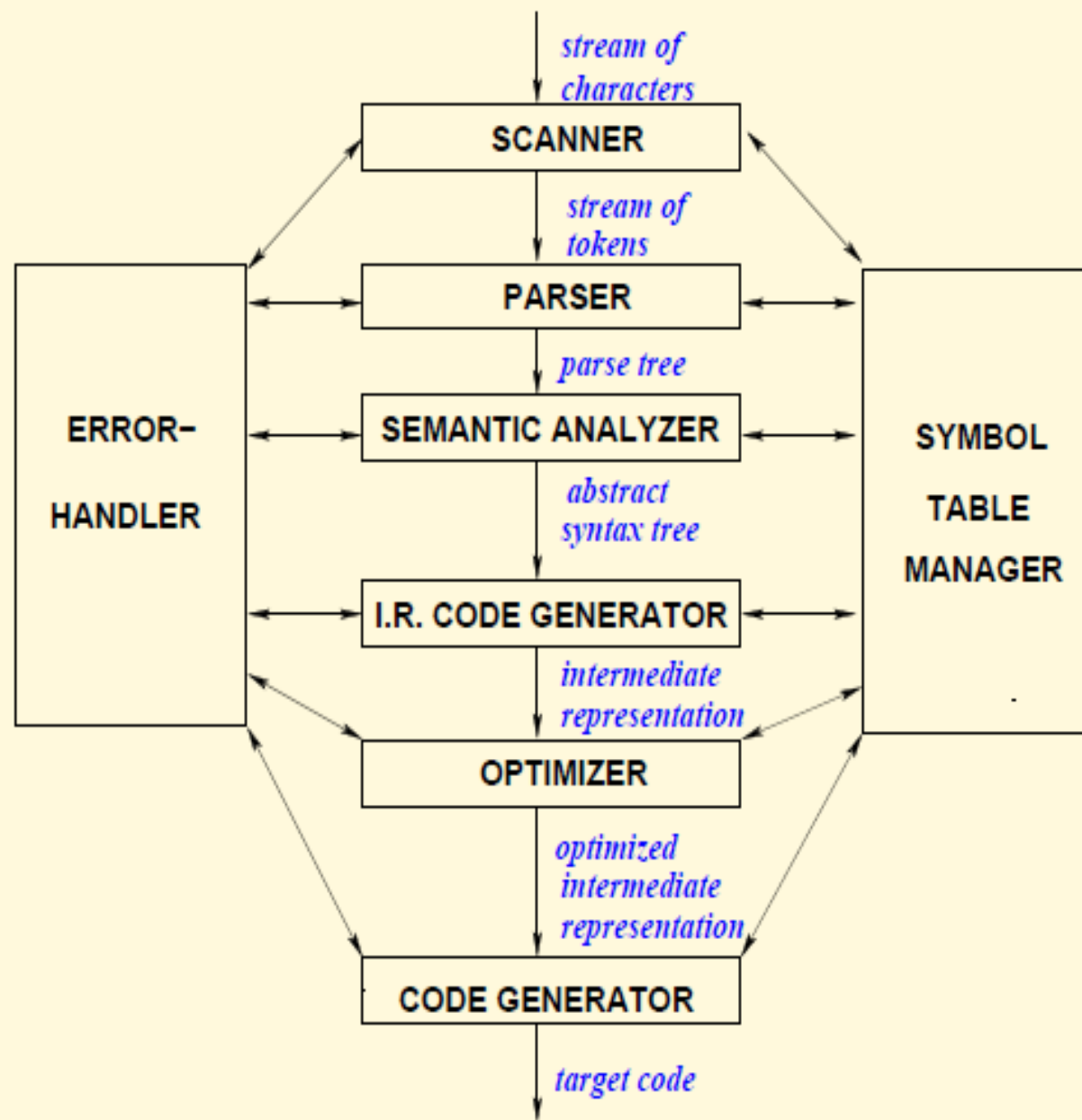


Semantic Analysis



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

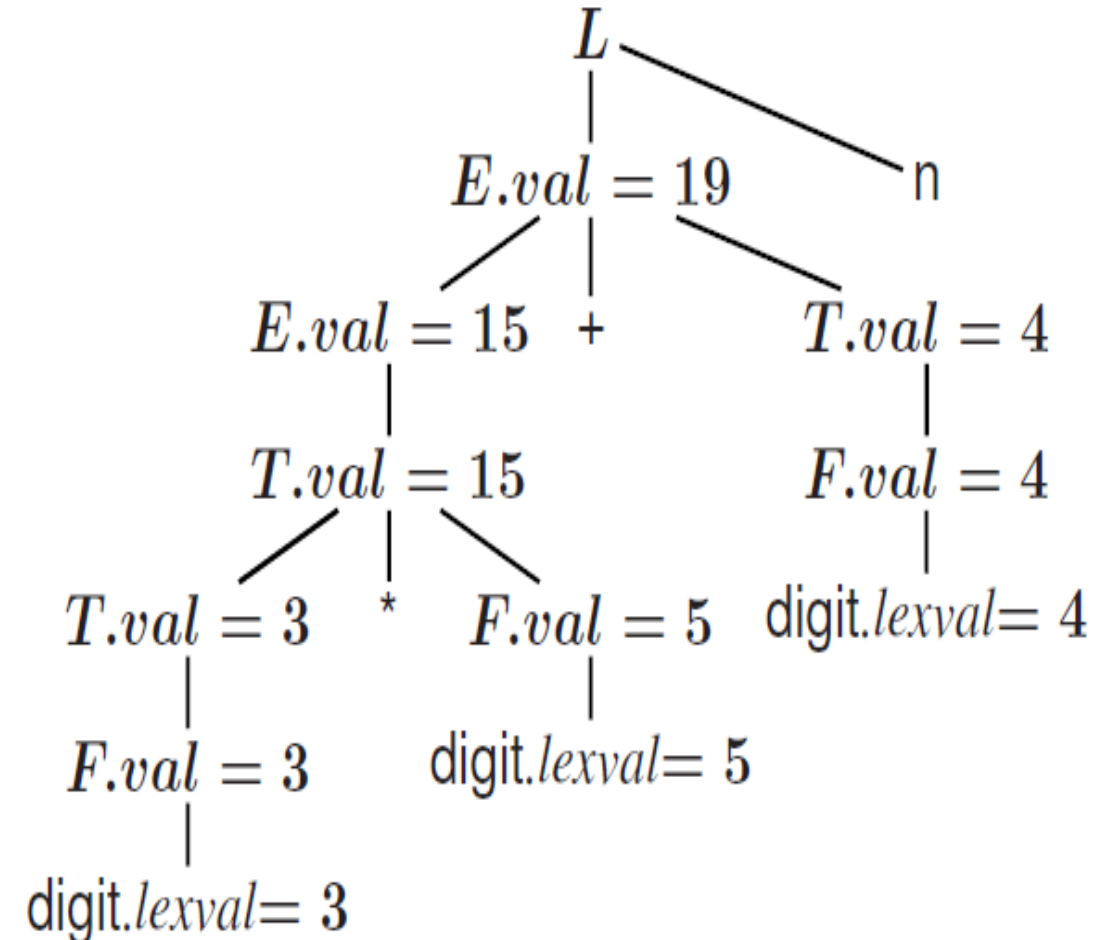
- To capture the context sensitive aspect, we need to understand
 - How to represent it (representation formalism)
 - How to implement it (implementation mechanism)
- As representation formalism we use Syntax Directed Translations
- Syntax Directed Translation relates an input sentence to its syntactic 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



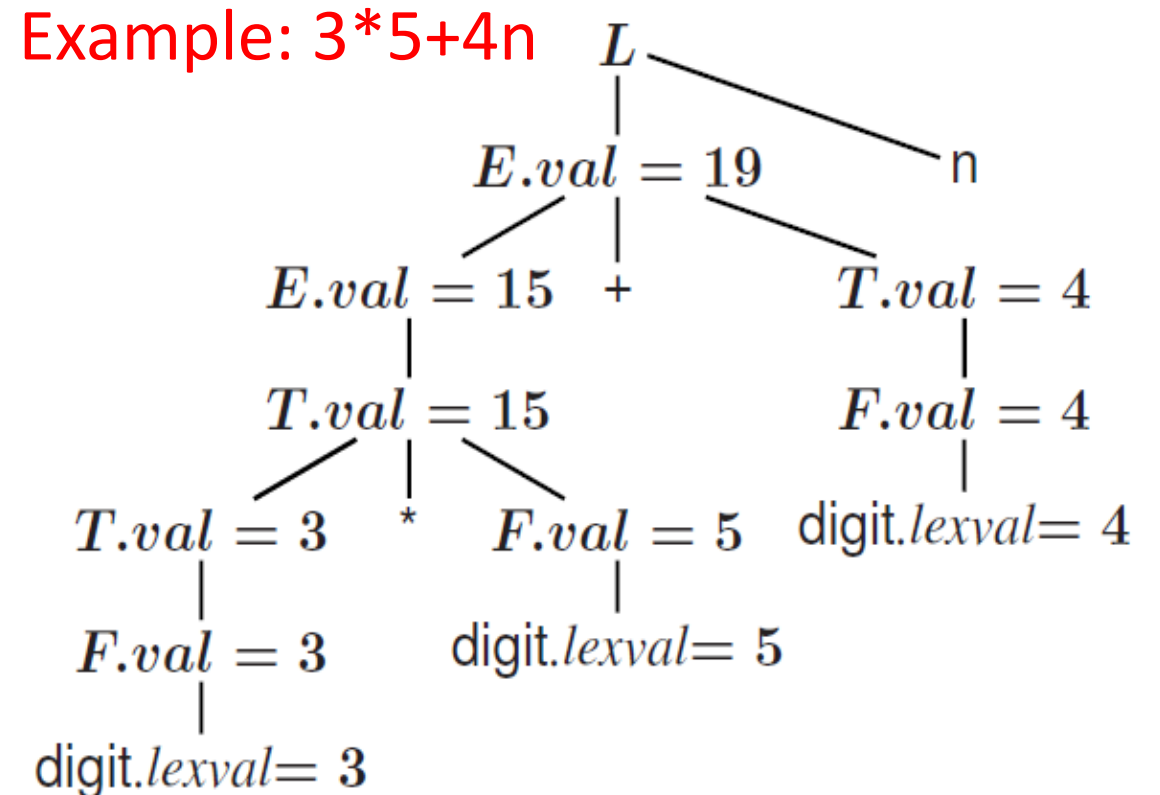
Syntax Directed Definitions

- 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
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$



L-Attributed Definitions

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

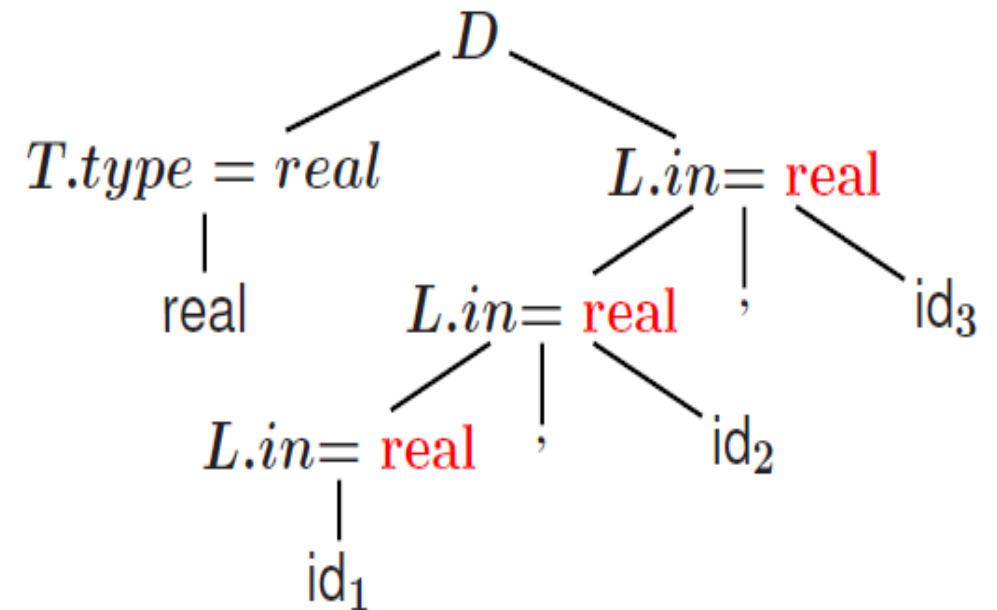
$$X_0 \rightarrow X_1 X_2 \dots X_n$$

- 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, \dots, 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
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in; \text{addtype}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.in)$



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 $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.

Extending a parser stack

<i>state</i>	<i>val</i>
<i>Z</i>	<i>Z.x</i>
<i>Y</i>	<i>Y.x</i>
<i>X</i>	<i>X.x</i>
...	...

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

INPUT	state	val	PRODUCTION USED
3*5+4 n	-	-	
*5+4 n	3	3	
*5+4 n	F	3	$F \rightarrow \text{digit}$
*5+4 n	T	3	$T \rightarrow F$
5+4 n	T *	3 _	
+4 n	T * 5	3 _ 5	
+4 n	T * F	3 _ 5	$F \rightarrow \text{digit}$
+4 n	T	15	$T \rightarrow T * F$
+4 n	E	15	$E \rightarrow T$
4 n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \rightarrow \text{digit}$
n	E + T	15 _ 4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	E n	19 _	
	L	19	$L \rightarrow E n$

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

Rules

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

$$S \rightarrow A1 A2 \{S.s = A1.s + A2.s\}$$
$$A \rightarrow a \{A.s = 1\}$$

- Synthesized and inherited attributes

$$S \rightarrow \{A1.in = 1; A2.in = 2\} A1 A2$$
$$A \rightarrow a \{A.s = 1\}$$

OR

$$S \rightarrow \{A1.in = 1\} A1 \{A2.in = 2\} A2$$
$$A \rightarrow a \{A.s = 1\}$$

Grammar:

$$S \rightarrow A A$$
$$A \rightarrow a$$

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

$$D \rightarrow T \{L.in := T.type\} L$$
$$T \rightarrow \text{int} \{T.type := \text{integer}\}$$
$$T \rightarrow \text{real} \{T.type := \text{real}\}$$
$$L \rightarrow \{L_1.in := L.in\} L_1, \text{id} \{addtype(\text{id.entry}, L.in)\}$$
$$L \rightarrow \text{id} \{addtype(\text{id.entry}, L.in)\}$$

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

Grammar: $S \rightarrow \text{while } (C) S$

$S \rightarrow \text{while } (C) S_1$ $L1 = \text{new}();$
 $L2 = \text{new}();$
 $S_1.\text{next} = L1;$
 $C.\text{false} = S.\text{next};$
 $C.\text{true} = L2;$
 $S.\text{code} = \text{label} \parallel L1 \parallel C.\text{code} \parallel \text{label} \parallel L2 \parallel S_1.\text{code}$

$S.\text{next}$: beginning of the next code

$S.\text{Code}$: code for S ends with a jump to $S.\text{next}$

$C.\text{true}$: beginning of the code to be executed if C is true

$C.\text{false}$: beginning of the code to be executed if C is false

$C.\text{code}$: code for C with a jump to either $C.\text{true}$ or $C.\text{false}$

Figure 5.27: SDD for while-statements

Implementation using Translation Scheme

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

Figure 5.27: SDD for while-statements

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

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 ($M \rightarrow \epsilon$).
- The semantic action is attached at the end of the production ($M \rightarrow \epsilon$).

Compile-Time Evaluation of Translation Schemes

Production	Semantic Rules
$S \rightarrow aAC$	$C.in = A.s$
$S \rightarrow bABC$	$C.in = A.s$
$C \rightarrow c$	$C.s = g(C.i)$
.....

Production	Semantic Rules
$S \rightarrow aAM_1C$	$M_1.in = A.s$ $C.in = M_1.s$
$S \rightarrow bABM_2C$	$M_2.in = A.s$ $C.in = M_2.s$
$C \rightarrow c$	$C.s = g(C.i)$
$M_1 \rightarrow \epsilon$	$M_1.s = M_1.in$
$M_2 \rightarrow \epsilon$	$M_2.s = M_2.in$
.....

Compile-Time Evaluation of Translation Schemes

- General rules to compute translations schemes during bottom-up parsing for an L-attributed grammar.
- For every production $A \rightarrow X_1 \dots X_n$ introduce n new markers M_1, \dots, M_n and replace the production by $A \rightarrow M_1 X_1 \dots M_n X_n$.
- Thus, the position of every synthesized and inherited attribute of X_j and A are known:
 - $X_j.s$ is stored in the *val* entry in the parser stack associated with X_j ;
 - $X_j.i$ is stored in the *val* entry in the parser stack associated with M_j ;
 - $A.i$ is stored in the *val* entry in the parser stack immediately before the position storing M_1

Compile-Time Evaluation of Translation Schemes

- Computing the inherited attribute $X_j.i$ after reducing $M_j \rightarrow \varepsilon$

$A.i$ is in $\text{val}[\text{top} - 2j + 2]$

$X_1.i$ is in $\text{val}[\text{top} - 2j + 3]$

$X_1.s$ is in $\text{val}[\text{top} - 2j + 4]$

$X_2.i$ is in $\text{val}[\text{top} - 2j + 5]$

And so on

	M_j	$X_j.i$
$top \rightarrow$	X_{j-1}	$X_{j-1}.s$
	M_{j-1}	$X_{j-1}.i$

	X_1	$X_1.s$
	M_1	$X_1.i$
$(top-2j+2) \rightarrow$	M_A	$A.i$
$(top-2j) \rightarrow$		

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.