Compiler Design

Samit Biswas

samit@cs.iiests.ac.in

Department of Computer Science and Technology,
Indian Institute of Engineering Science and Technology, Shibpur

September 24, 2018

Syntax Directed Translation

Syntax Directed Translation

- We associate information with the programming language constructs by attaching attributes to grammar symbols.
- Values of this attributes are evaluated by semantic rules associated with the production rules.

Syntax Directed Translation

- We associate information with the programming language constructs by attaching attributes to grammar symbols.
- Values of this attributes are evaluated by semantic rules associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate code.
  - may put information into the symbol table.
  - may perform type checking.
- An attribute may hold almost anything.
  - a string, a number, a memory location, a complex record

Syntax Directed Definitions and Translation schemes

when we associate semantic rules with productions, we use two notations:

- Syntax Directed Definition.
- Translation schemes.

Syntax Directed Definitions and Translation schemes
when we associate semantic rules with productions, we use two notations:

- Syntax Directed Definition.
- Translation schemes.

Syntax Directed Definition:

- give high level specification for the translation.
- hide many implementation details such as order of evaluation of semantic actions.
- we associate production rules with a set of semantic actions, and we do not say when they will be evaluate.

Syntax Directed Definitions and Translation schemes
when we associate semantic rules with productions, we use two notations:

- Syntax Directed Definition.
- Translation schemes.

Syntax Directed Definition:

- give high level specification for the translation.
- hide many implementation details such as order of evaluation of semantic actions.
- we associate production rules with a set of semantic actions, and we do not say when they will be evaluate.

Translation schemes:

- indicate the order of evaluation of semantic actions associated with a production rule.

Syntax Directed Translation

Conceptually with both Syntax Directed Translation and Translation Scheme

- Parse the input token stream.
- Build the parse tree.
- Traverse the tree to evaluate the semantic rules at the parse tree nodes.

Input Stream $\longrightarrow$ Parse Tree $\longrightarrow$ dependency Graph $\longrightarrow$ evaluation order for semantic rules

Syntax Directed Definitions

- A syntax directed definition is a generalization of Context Free Grammar in which:
  - Each grammar symbol is associated with a set of attribute.
  - This set of attributes can be classified into two:
    - Synthesized Attributes.
    - Inherited Attributes.
  - Each production rule is associated with a set of semantic rules.

Syntax Directed Definitions

- A syntax directed definition is a generalization of Context Free Grammar in which:
  - Each grammar symbol is associated with a set of attribute.
  - This set of attributes can be classified into two:
    - Synthesized Attributes.
    - Inherited Attributes.
  - Each production rule is associated with a set of semantic rules.
- The value of an attribute at a parse tree node is defined by the semantic rule associated with a production at that node.
- The value of a **Synthesized Attribute** at a node is computed from the values of attributes at the children in that node of the parse tree.
- The value of a **Inherited Attribute** at a node is computed from the values of attributes at the sibling and parents in that node of the parse tree.

Syntax Directed Definitions

Examples:

- Synthesized Attributes:
  $E \rightarrow E_1 + E_2 \qquad \{E.val = E_1.val + E_2.val\}$
- Inherited Attributes:
  $A \rightarrow XYZ \qquad \{Y.val = 2 * A.val\}$

Syntax Directed Definitions

Examples:

- Synthesized Attributes:
  $E \rightarrow E_1 + E_2 \qquad \{E.val = E_1.val + E_2.val\}$
- Inherited Attributes:
  $A \rightarrow XYZ \qquad \{Y.val = 2 * A.val\}$

- Semantic rules setup and dependencies between attributes which can be represented by a dependency graph.
- Dependency graph determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. A semantic rule may also have some side effects such as printing a value.

.

Annotated Parse tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

Annotated Parse tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree.**
- Values of attributes in nodes of annotated parse tree are either :
  - initialized to constant values or by the lexical analyzer.
  - determined by the semantic rules.

Annotated Parse tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree.**
- Values of attributes in nodes of annotated parse tree are either :
  - initialized to constant values or by the lexical analyzer.
  - determined by the semantic rules.
- The process of computing the attributes values at the nodes is called **annotating** or **decorating** of the parse tree.
- The order of these computations depends on the **dependency graph** induced by the semantic rules.

Syntax-Directed Definition

In a Syntax-Directed Definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form

$$b = f(c_1, c_2, c_3, \ldots, c_k)$$

where f is a function and b can be one of the following:

- b is a **synthesized attribute** of A and $c_1, c_2, c_3, \ldots, c_k$ are attributes of the grammar symbols in α, or
- b is an **inherited attribute** of one of the grammar symbols on the right side of the production $A \rightarrow \alpha$ and $c_1, c_2, c_3, \ldots, c_k$ are attributes of the grammar symbols in $\{A, \alpha\}$.

Example with Synthesized attributes

- Grammar Symbols: $L, E, T, F, n, +, *, (,), digit$
- Non-Terminal : L, E, T, F have an attribute called *val*.
- Terminal *digit* have an attribute called *lexval*.
- The value for *lexval* is provided by the lexical analyser.

Table: Syntax-directed Definition

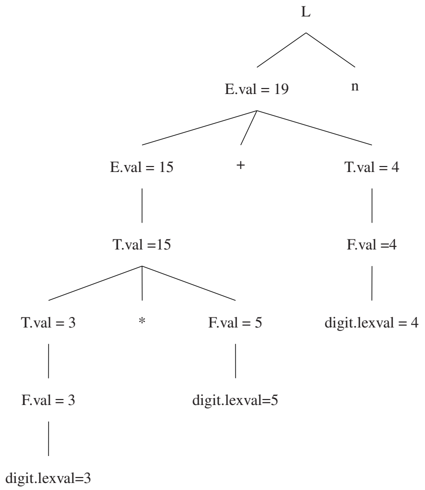
Productions	Semantic Rules
$L \rightarrow En$	print("E.val");
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

Draw the tree (Annotated parse Tree)

(Example: $3 * 5 + 4n$)

Draw the tree (Annotated parse Tree)

(Example: $3 * 5 + 4n$)



Example with Inherited attributes

A declaration generated by the non-terminal D in the Syntax directed Definition consists of keyword *int* or *real* followed by a list of Identifiers.

Table: Syntax-directed Definition

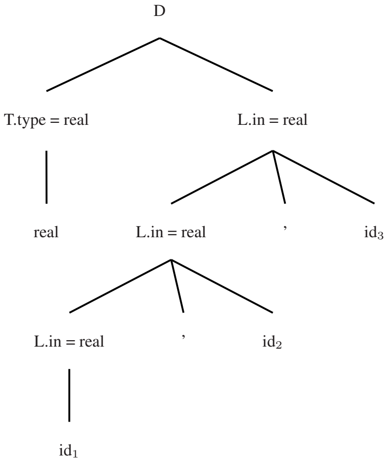
Productions	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

Draw the tree (Annotated parse Tree)

(Example: $real\ id_1, id_2, id_3$)

Draw the tree (Annotated parse Tree)

(Example: *real* id_1, id_2, id_3)



Dependency Graph

- Directed Graph.
- Shows Intermediate dependencies between attributes.
- Construction
  - Put each semantic rule into the form $b = f(c_1, \ldots, c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a **procedure call**.
  - Eg.

$L \rightarrow En$	print("E.val")
Becomes:	dummy = print("'E.val')
etc.	

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.

Dependency Graph Construction

- **for** each node n in the parse tree **do**
  for each attribute a of the grammar symbol at node a **do**
  - construct a node in the dependency graph for a;

Dependency Graph Construction

- **for** each node n in the parse tree **do**
  for each attribute a of the grammar symbol at node a **do**
    - construct a node in the dependency graph for a;
- **for** each node n in the parse tree **do**
  - for each semantic rule $b = f(c_1, c_2, \ldots, c_k)$ associated with the production used at n **do**
    - **for** i = 1 to k do
      Construct an edge from the node for c_i to the node for b;

Example with Synthesized attributes

- Grammar Symbols: $L, E, T, F, n, +, *, (,), digit$
- Non-Terminal : L, E, T, F have an attribute called *val*.
- Terminal *digit* have an attribute called *lexval*.
- The value for *lexval* is provided by the lexical analyser.

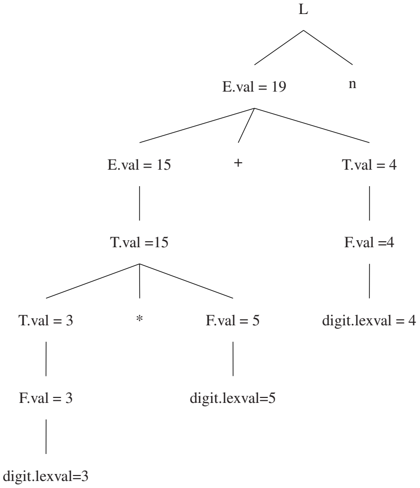
Subscript is E1 is to differentiate it from E in the semantic rules

Table: Syntax-directed Definition

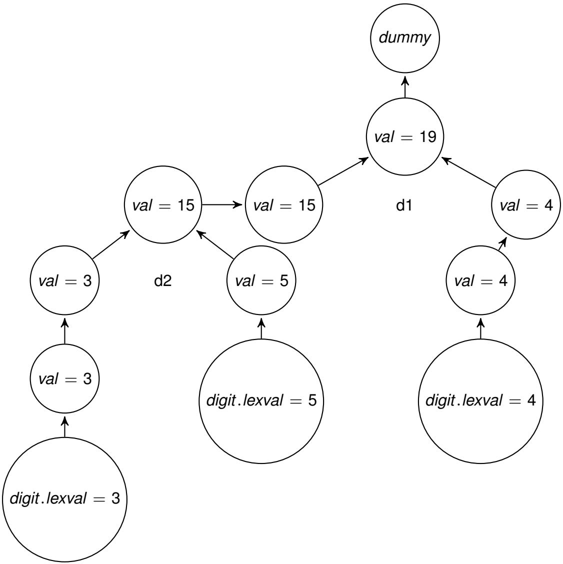
Productions	Semantic Rules
$L \rightarrow En$	print("E.val");
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

(Annotated parse Tree)

(Example: $3 * 5 + 4n$)



Example: Dependency Graph



Example with Inherited attributes

A declaration generated by the non-terminal *D* in the Syntax directed Definition consists of keyword *int* or *real* followed by a list of Identifiers.

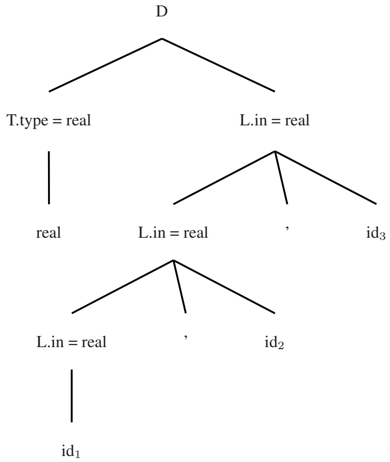
Table: Syntax-directed Definition

Productions	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

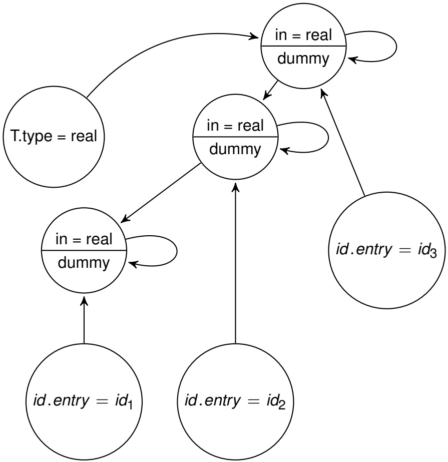
L.in -> L.inh (inherited)

Annotated parse Tree

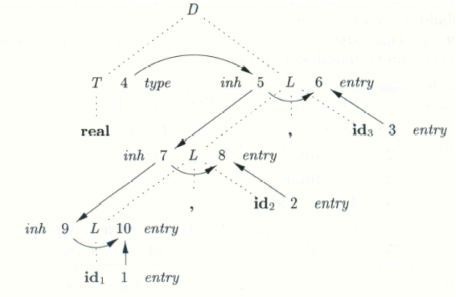
(Example: *real* id_1, id_2, id_3)



Example: Dependency Graph



Example: Evaluation Order



- $a_4 = real$
- $a_5 = a_4$
- $addtype(id_3.entry, a_5)$
- $a_7 = a_5$
- $addtype(id_2.entry, a_7)$
- $a_9 = a_7$
- $addtype(id_1.entry, a_9)$

Evaluation Order of Semantic Rules

Several methods have been proposed for the evaluation of semantic rules.

- Parse Tree Method:
  - At compile time evaluation order obtained from dependency graph constructed from the parse tree.
  - Fails if dependency graph contains a cycle.
- Rule Based methods:
  - Semantic rules analyzed by hand or specialized tools at compiler construction time.
  - Order of evaluation of attributes associated with a production is pre-determined at compiler construction time.
- Oblivious methods:
  - Evaluation order is chosen without considering the semantic rules.
  - Restricts the class of syntax directed definitions that can be implemented.
  - Order of evaluation is forced by parsing method.

Construction of Syntax tree

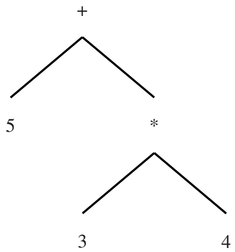
Syntax - Tree

- an intermediate representation of the **compiler's input**.
- A condensed form of parse tree.
- Syntax tree shows the syntactic structure of the programme while omitting the irrelevant details.
- Operators or keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.

Syntax directed translation can be based on syntax tree as well as parse tree.

Syntax - Tree Example

5 + 3 * 4



- Leaves: identifiers or constants.
- Internal nodes: Labelled with operations.
- Children of a node are its operands.

Constructing Syntax trees for an Expression:

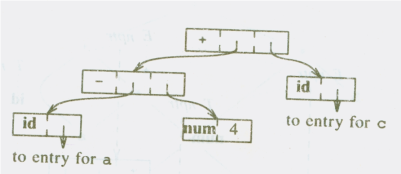
- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called label of the node) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.
- Functions used to create nodes of syntax tree for expressions with binary operator are given below
  - **mknode**(op, left, right)
  - **mkleaf**(id, entry)
  - **makeleaf** (num, val)

  Each function returns a pointer to a newly created node.

Example

a - 4 + c

1. $p_1 = mkleaf(id, entrya);$
2. $p_2 = mkleaf(num, 4);$
3. $p_3 = mknode('-', p_1, p_2);$
4. $p_4 = mkleaf(id, entryc);$
5. $p_5 = mknode('+', p_3, p_4);$

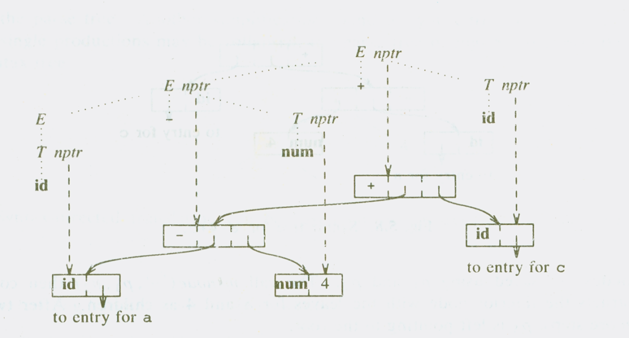


Syntax-Directed definition for Constructing Syntax Trees

Table: Syntax-directed definition for constructing a syntax tree.

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.nptr = mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow id$	$T.nptr = mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr = mkleaf(num, num.val)$

Example: Construction of a Syntax-tree for a-4+c



Directed Acyclic Graphs for Expression

Bottom up Evaluation of S - Attributed Definitions

- A translator for an S-attributed definition can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- We put the values of the synthesized attributes of the grammar symbols a stack that has extra fields to hold the values of attributes.
-

Table: Implementation of a Calculator with an LR parser

Production	Code Fragment
$L \rightarrow En$	print(val[top]);
$E \rightarrow E_1 + T$	$val[ntop] = val[top - 2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntop] = val[top - 2] * val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntop] = val[top - 1]$
$F \rightarrow digit$	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$
*5+4n	T	3	$T \rightarrow F$
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	F * F	3 _ 5	$F \rightarrow digit$
+4n	T	15	$T \rightarrow T * F$
+4n	E	15	$E \rightarrow T$
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \rightarrow digit$
n	E + T	15 _ 4	$T \rightarrow F$
n	E	19	$T \rightarrow F$
En	E n	19 _	
	L	19	$L \rightarrow En$

Bottom-Up Evaluation of Inherited Attributes

Type Checking

Type Checking

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- This generally means that all operands in any expression are of appropriate types and numbers.
- Mostly what we do in semantic analysis phase is type checking.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.
- If a problem found, e.g. one tries to add a character to a double in C, we encounter a type error.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.
- If a problem found, e.g. one tries to add a character to a double in C, we encounter a type error.
- A language is considered ***strongly-typed*** if each and every type error is detected during compilation.
- Type checking can be done in compile time or in execution time.

Static Type Checking

- Static type checking is done at compile time. The information type checker needs is obtained via declarations and stored in a master symbol table.
- After this information is collected, the types involved in each operation are checked.

Static Type Checking

- Static type checking is done at compile time. The information type checker needs is obtained via declarations and stored in a master symbol table.
- After this information is collected, the types involved in each operation are checked.
- For example, if a and b are of type int and we assign very large values to them, $a * b$ may not be in the the acceptable range of ints, or an attempt to compute the ratio between two integers may raise a division by zero. These kind of type errors usually can not be detected at compile time.

Dynamic Type Checking

- Dynamic type checking is implemented by including type information for each data location at runtime.
- For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type".
- The execution of any operation begins by first checking these type tags. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution.

Type Expressions

The type of a language construct will be denoted by a “type expression”.

The few basic type expressions are as follows:

- The basic types are boolean, char, integer, and real. A special basic type, type_error, will signal an error during type checking. Finally, a basic type void denoting “the absence of a value ” allows statements to be checked.
- Type expression may be named, a type name is a type expression.

Type Expressions

The type of a language construct will be denoted by a “type expression”.

The few basic type expressions are as follows:

- The basic types are boolean, char, integer, and real. A special basic type, type_error, will signal an error during type checking. Finally, a basic type void denoting “the absence of a value ” allows statements to be checked.
- Type expression may be named, a type name is a type expression.
- A type constructor applied to type expression is a type expression. Constructors include:
    - Arrays
    - Products
    - Records
    - Pointers
    - Functions

Arrays:
If T is a Type expression, then array(I, T) is a type expression denoting the type of an array with elements of type *T* and index set *I*. *I* is often a range of integers. For example

- var A: array[1..10] of integer;

Associates the type expression: **array(1..10, integer) with A**

Products
If T_1 and T_2 are type expressions, their Cartesian product $T_1 \times T_2$ is a type expression.

Records

The record type constructor will be applied to a tuple formed from field names and field types.

type	row =	record	
		address:	integer;
		lexeme:	array [1..15] of char
		end ;	
var	table:	array[1..10] of row;	

declares the name row representing the type expression **record((address** $\times$ **integer)** $\times$ **(lexeme** $\times$ **array(1 .. 15, char)))**

The variable *table* to be an array of records of this type.

Pointers:

If T is a type expression, then $pointer(T)$ is a type expression denoting the type “pointer to an object of type T”.

For example,

var p: $\uparrow$ row

declares variable p to have type pointer (row).

Functions:
Mathematically, a function maps elements of one set, the domain, to another set, the range. We may treat functions in programming languages as mapping a *domain type* D to a *range type* R. The type of such a function will be denoted by $D \to R$.
As for example,

function f(a, b : char): ↑ integer;

The type of f is denoted by the type expression

$$char \times char \to pointer(integer)$$

Specification of a simple Type Checker

The following grammar generate programs, represented by the nonterminal P, consisting of a sequence of declarations D followed by a single expression E.

$P \rightarrow D; E$

$D \rightarrow D; D | id : T$

$T \rightarrow char | integer | array[num] of\ T | \uparrow T$

$E \rightarrow literal\ | num |\ id |\ EmodE\ | E[E]\ | E \uparrow$

Table: Translation Scheme that saves the type of an identifier

Productions	Associated rules for type
$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow id : T$	{ addtype(id.entry, T.type)}
$T \rightarrow char$	{ T.type = char }
$T \rightarrow integer$	{ T.type = integer }
$T \rightarrow \uparrow T_1$	{ T.type = pointer($T_{1.type}$) }
$T \rightarrow array[num]\ of\ T_1$	{ T.type = array($1 \ldots num.val,\ T_{1.type}$ }

Type Checking of Expressions

Table: Associated rules for Type Checking

Productions	Associated rules for type
$E \rightarrow literal$	E.type = char
$E \rightarrow num$	E.type = integer
$E \rightarrow id$	E.type = lookup(id.entry)
$E \rightarrow E_1\, mod E_2$	E.type = **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then** *integer* **else** *type_error*
$E \rightarrow E_1[E_2]$	E.type = **if** $E_{2.type} = integer$ **and** $E_{1.type} = array(s, t)$ **then** *t* **else** *type_error*
$E \rightarrow E_1 \uparrow$	E.type = if $E_{1.type}$=pointer(t) then t **else** *type_error*

Type Checking of Statements

The state statements we consider are assignment, conditional, and while statements.

Table: default

Productions	Associated rules for type
$S \rightarrow id = E$	{S.type = **if** id.type == E.type **then** void **else** *type_error* }
$S \rightarrow$ *if* E then S_1	{S.type = **if** E.type == Boolean **then** $S_1.type$ **else** *type_error* }
$S \rightarrow$ *while* E do S_1	{S.type = **if** E.type == Boolean **then** $S_1.type$ **else** *type_error* }
$S \rightarrow S_1 ; S_2$	{S.type = **if** S_1.type == void **and** S_2.type == void **then** void **else** *type_error* }

Type Checking of Functions

Productions	Associated actions
$T \rightarrow T_1 \ ' \rightarrow' \ T_2$	$\{T.type = T_1.type \rightarrow T_2.type\ \}$
$E \rightarrow E_1(E_2)$	{E.type = if $E_2.type$ == s and E_1.type == $s \rightarrow t$ **then** t **else** *type_error* }