**Syntax Directed Translation Scheme**

**Attributes:**

**Variables:**

VAR\_DEFINITION\_TAG.type = type of variable (string) synthesized

TYPE\_INDICATOR.category = category of variable, can be: none, basic, array or pointer (string) synthesized

TYPE\_INDICATOR.type = type of variable (string) synthesized

TYPE\_INDICATOR.size = integer value of lexeme, calculated only it it is an array (integer) synthesized

BASIC\_TYPE.type = type of variable (string) synthesized synthesized

ARRAY\_TYPE.type = type of variable (string) synthesized

ARRAY\_TYPE.size = integer value of lexeme (integer) synthesized

POINTER\_TYPE.type = type of variable (string) synthesized

POINTER\_TYPE\_TAG.type = type of variable (string) synthesized

SIZE.type = type of variable (string) synthesized

SIZE.val = integer value of lexeme (integer) synthesized

COMMAND.category = category of variable, can be: none, basic, array or pointer (string) synthesized

COMMAND.type = type of variable (string) synthesized

RECEIVER.category = category of variable, can be: none, basic, array or pointer (string) synthesized

RECEIVER.type = type of variable (string) synthesized

RECEIVER.role = identify if variable is actually a variable or user\_defined\_type (string) inherited

RECEIVER\_TAG.type = type of variable (string) synthesized

RECEIVER\_TAG.code = code represented by variable (string) synthesized

EXPRESSION.category = category of variable, can be: none, basic, array or pointer (string) synthesized

EXPRESSION.type = type of variable (string) synthesized

EXPRESSION.val = value of expression (string) synthesized

EXPRESSION\_TAG.type = type of variable (string) synthesized

EXPRESSION\_TAG.code = code represented by variable (string) synthesized

**Terminals (tokes):**

id.lexeme = the actual name in text of id (string) synthesized synthesized

id.role = identify if id is actually a variable or user\_defined\_type (string) synthesized

id.category = category of id, can be: none, basic, array or pointer (string) synthesized

id.type = type of id (string) synthesized

type\_name.lexeme = the actual name in text of type\_name (string) synthesized

type\_name.role = identify if type\_name is actually a variable or user\_defined\_type (string) synthesized

type\_name.category = category of type\_name, can be: none, basic, array or pointer (string) synthesized

type\_name.type = type of type\_name (string) synthesized

|  |  |  |  |
| --- | --- | --- | --- |
|  | **#** | **Derivation rule** | **Semantic action** |
|  | 1 | PROGRAM 🡪 BLOCK |  |
|  | 2 | BLOCK 🡪 **block** DEFINITIONS ; **begin** COMMANDS ; **end** | cur\_table = make\_table (cur\_table)  ...  cur\_table = pop\_table (cur\_table) |
|  | 3 | DEFINITIONS 🡪 DEFINITION DEFINITIONS\_TAG |  |
|  | 4 | DEFINITIONS\_TAG 🡪 |  |
|  | 5 | DEFINITIONS\_TAG 🡪 ; DEFINITION DEFINITIONS\_TAG |  |
|  | 6 | DEFINITION 🡪 VAR\_DEFINITION |  |
|  | 7 | DEFINITION 🡪 TYPE\_DEFINITION |  |
|  | 8 | VAR\_DEFINITION 🡪 id : VAR\_DEFINITION\_TAG | id\_entry = lookup(cur\_table, id. lexeme); // lookup performs search **current scope ONLY**  if (id\_entry != NULL){ // id already been declared  printf (“ERROR: %s already been declared in this scope \n”, id. lexeme);  } else { // id does not declared - we can save it safely  id. role = "variable"  id.type = VAR\_DEFINITION\_TAG.type  set\_id(cur\_table, id);  } |
|  | 9 | VAR\_DEFINITION\_TAG 🡪 BASIC\_TYPE | VAR\_DEFINITION\_TAG.type = BASIC\_TYPE.type |
|  | 10 | VAR\_DEFINITION\_TAG 🡪 type\_name | id\_table\_entry = find(cur\_table, type\_name.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "user\_defined\_type") { // is a type\_name  VAR\_DEFINITION\_TAG.type = type\_name.lexeme  } else { // not a type  VAR\_DEFINITION\_TAG.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, type\_name.lexeme);  }  } else {  VAR\_DEFINITION\_TAG.type = "error type";  printf(“ERROR: % s is not declared\ n”, type\_name.lexeme);  } |
|  | 11 | TYPE\_DEFINITION 🡪 **type** type\_name **is** TYPE\_INDICATOR | type\_name\_entry = lookup(cur\_table, type\_name. lexeme); // lookup performs search **current scope ONLY**  if (type\_name\_entry != NULL){ // type\_name already been declared  printf (“ERROR: %s already been declared in this scope \n”, type\_name. lexeme);  } else { // type\_name does not declared - we can save it safely  type\_name.type = TYPE\_INDICATOR.type  type\_name. category = TYPE\_INDICATOR. category  set\_id(cur\_table, type\_name);  } |
|  | 12 | TYPE\_INDICATOR 🡪 BASIC\_TYPE | TYPE\_INDICATOR.type = BASIC\_TYPE.type  TYPE\_INDICATOR. category = "basic" |
|  | 13 | TYPE\_INDICATOR 🡪 ARRAY\_TYPE | TYPE\_INDICATOR.type = ARRAY\_TYPE.type  TYPE\_INDICATOR.category = "array"  TYPE\_INDICATOR.size = ARRAY\_TYPE.size |
|  | 14 | TYPE\_INDICATOR 🡪 POINTER\_TYPE | TYPE\_INDICATOR.type = POINTER\_TYPE.type  TYPE\_INDICATOR. category = "pointer" |
|  | 15 | BASIC\_TYPE 🡪 **integer** | BASIC\_TYPE.type = "integer" |
|  | 16 | BASIC\_TYPE 🡪 **real** | BASIC\_TYPE.type = "real" |
|  | 17 | ARRAY\_TYPE 🡪 **array** [ SIZE ] **of** BASIC\_TYPE | ARRAY\_TYPE.type = BASIC\_TYPE.type;  ARRAY\_TYPE.size = SIZE.val; |
|  | 18 | POINTER\_TYPE 🡪 ^ POINTER\_TYPE\_TAG | POINTER\_TYPE.type = POINTER\_TYPE\_TAG.type |
|  | 19 | POINTER\_TYPE\_TAG 🡪 BASIC\_TYPE | POINTER\_TYPE\_TAG.type = BASIC\_TYPE.type |
|  | 20 | POINTER\_TYPE\_TAG 🡪 type\_name | id\_table\_entry = find(cur\_table, type\_name.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "user\_defined\_type") { // is a type\_name  POINTER\_TYPE\_TAG.type = type\_name.lexeme  } else { // not a type  POINTER\_TYPE\_TAG.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  POINTER\_TYPE\_TAG.type = "error type";  printf(“ERROR: % s is not declared\ n”, type\_name.lexeme);  } |
|  | 21 | SIZE 🡪 int\_num | SIZE.val = string\_to\_num(int\_num. lexeme)  SIZE.type = "integer" |
|  | 22 | COMMANDS 🡪 COMMAND COMMANDS\_TAG |  |
|  | 23 | COMMANDS\_TAG 🡪 |  |
|  | 24 | COMMANDS\_TAG 🡪 ; COMMAND COMMANDS\_TAG |  |
|  | 25 | COMMAND 🡪 RECEIVER = EXPRESSION | if(RECEIVER. category == EXPRESSION. category and RECEIVER.type == EXPRESSION.type){  ...  } else {  COMMAND.type = "error type";  printf (“ERROR: left side of assignment is not the same as right side. \n”);  }  if(RECEIVER. role == "user\_defined\_type"){  if(RECEIVER. category == "basic" and EXPRESSION. category == "basic"){  if(basic\_type (RECEIVER.type) =="integer" and basic\_type(EXPRESSION.type) == "integer"){  ...  } if(basic\_type (RECEIVER.type) =="real" and basic\_type(EXPRESSION.type) == "real"){  ...  } else {  COMMAND.type = "error type";  printf (“ERROR: left side of assignment is not the same basic type as right side. \n”);  }  } if (RECEIVER. category == "pointer" and EXPRESSION. category =="pointer"){  if(RECEIVER.type == EXPRESSION.type){  ...  } else {  COMMAND.type = "error type";  printf (“ERROR: wrong assignment. \n”);  }  } if (RECEIVER. category == "array"){  COMMAND.type = "error type";  printf (“ERROR: assignment to array is forbidden. \n”);  }  } |
|  | 26 | COMMAND 🡪 **when**  ( EXPRESSION rel\_op EXPRESSION )  **do**  COMMANDS ;  **default**  COMMANDS ;  **end\_when** | // no semantic errors according to assignment |
|  | 27 | COMMAND 🡪 **for**  ( id = EXPRESSION ; id rel\_op EXPRESSION ; id ++ )  COMMANDS ;  **end\_for** | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // not a type\_name  if (id.category == "basic") { // check for (id ++)  if (EXPRESSION.type == id.type) { // id = EXPRESSION  if (EXPRESSION.category == "basic") { // id rel\_op EXPRESSION  ...  } else {  COMMAND.type = "error type";  printf(“ERROR: arithmetic expressions cannot be applied to arguments of array and pointer”);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: left and right sides must be of exactly same type”);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: arithmetic expressions cannot be applied to arguments of array and pointer”);  }  } else { // is a type  COMMAND.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: % s is not declared\ n”, id.lexeme);  } |
|  | 28 | COMMAND 🡪 id = **malloc** ( **size\_of** ( type\_name ) ) | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  type\_table\_entry = find(cur\_table, type\_name.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (type\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // meaning not a type\_name  if (type\_table\_entry.role == "user\_defined\_type") { // meaning a type\_name  if (id.category == "pointer" and id.type == type\_name.lexeme) {  COMMAND.category = id.category;  COMMAND.type = id.type;  } else {  COMMAND.type = "error type";  printf(“ERROR: % s is not pointer to % s\ n”, id.name, type\_name.lexeme);  }  } else { // role is variable  COMMAND.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, type\_name.lexeme);  }  } else { // role is user\_defined\_type  COMMAND.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: % s is not declared\ n”, type\_table\_entry.lexeme);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: % s is not declared\ n”, id\_table\_entry.lexeme);  } |
|  | 29 | COMMAND 🡪 **free** ( id ) | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // not a type\_name  ...  } else { // is a type  COMMAND.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  COMMAND.type = "error type";  printf(“ERROR: % s is not declared\ n”, id.lexeme);  } |
|  | 30 | COMMAND 🡪 BLOCK |  |
|  | 31 | RECEIVER 🡪 id RECEIVER\_TAG | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // not a type\_name  if (RECEIVER\_TAG.code != "^") {  if (RECEIVER\_TAG.code[0] != "[") {  ...  } else {  if (id.category == "array"  and RECEIVER\_TAG.type == "integer") {  RECEIVER.type = id.type;  } else {  RECEIVER.type = "error type";  printf(“ERROR\ n”);  }  }  } else {  if (id.category == "pointer") {  RECEIVER.type = id.type;  } else {  RECEIVER.type = "error type";  printf(“ERROR\ n”);  }  }  } else { // is a type  RECEIVER.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  RECEIVER.type = "error type";  printf(“ERROR: % s is not declared\ n”, id.lexeme);  } |
|  | 32 | RECEIVER\_TAG 🡪 |  |
|  | 33 | RECEIVER\_TAG 🡪 [ EXPRESSION ] | RECEIVER\_TAG.type = EXPRESSION.type;  RECEIVER\_TAG.code = "["|| EXPRESSION.code || "]"; |
|  | 34 | RECEIVER\_TAG 🡪 ^ | RECEIVER\_TAG.code = "^";  RECEIVER\_TAG.type = "pointer"; |
|  | 35 | EXPRESSION 🡪 int\_num | EXPRESSION.val = string\_to\_num(int\_num. lexeme)  EXPRESSION.type = "integer" |
|  | 36 | EXPRESSION 🡪 real\_num | EXPRESSION.val = string\_to\_num(real \_num. lexeme)  EXPRESSION.type = "real" |
|  | 37 | EXPRESSION 🡪 & id | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // not a type\_name  EXPRESSION.category = "pointer";  EXPRESSION.type = id.type;  } else { // is a type  EXPRESSION.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  EXPRESSION.type = "error type";  printf(“ERROR: % s is not declared\ n”, id.lexeme);  } |
|  | 38 | EXPRESSION 🡪 **size\_of** ( type\_name ) | id\_table\_entry = find (cur\_table, type\_name. lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "user\_defined\_type") { // is a type\_name  EXPRESSION.type = "integer";  } else { // not a type  EXPRESSION.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  EXPRESSION.type = "error type";  printf (“ERROR: %s is not declared \n”, type\_name. lexeme);  } |
|  | 39 | EXPRESSION 🡪 id EXPRESSION\_TAG | id\_table\_entry = find(cur\_table, id.lexeme); // find performs hierarchical search  if (id\_table\_entry != NULL) { // if true then id was declared  if (id\_table\_entry.role == "variable") { // meaning not a type\_name  if (EXPRESSION\_TAG.code != "^") {  if (EXPRESSION\_TAG.code[0] != "[") {  if (EXPRESSION\_TAG.code[0] != SOME\_AR\_OP) {  ...  } else {  if (  (id.type == EXPRESSION\_TAG.type) and  (id.category == "basic" and EXPRESSION\_TAG.category == "basic")  ) {  EXPRESSION.type = id.type;  } else {  EXPRESSION.type = "error type";  printf(“ERROR: arithmetic expressions cannot be applied to arguments of array and pointer \ n”);  }  }  } else {  if (id.category == "array" and EXPRESSION\_TAG.type == "integer") {  EXPRESSION.type = id.type;  } else {  EXPRESSION.type = "error type";  printf(“ERROR\ n”);  }  }  } else {  if (id.category == "pointer") {  EXPRESSION.type = id.type;  } else {  EXPRESSION.type = "error type";  printf(“ERROR\ n”);  }  }  } else { // role is user\_defined\_type  EXPRESSION.type = "error type";  printf(“ERROR\ % s can not be a type.\n”, id.lexeme);  }  } else {  EXPRESSION.type = "error type";  printf(“ERROR: % s is not declared\ n”, id.lexeme);  } |
|  | 40 | EXPRESSION\_TAG 🡪 |  |
|  | 41 | EXPRESSION\_TAG 🡪 [ EXPRESSION ] | EXPRESSION\_ TAG.type = EXPRESSION.type;  EXPRESSION\_ TAG.code = "["|| EXPRESSION.code || "]"; |
|  | 42 | EXPRESSION\_TAG 🡪 ^ | EXPRESSION.code = "^";  EXPRESSION.type = "pointer"; |
|  | 43 | EXPRESSION\_TAG 🡪 ar\_op EXPRESSION | EXPRESSION\_ TAG.type = EXPRESSION.type;  EXPRESSION\_ TAG.code = ar\_op. lexeme || EXPRESSION.code; |