

CSE 131A

Lecture 16

Type conversion

More on evaluation environments

Scott B. Baden

Department of Computer Science & Engineering

University of California, San Diego

Winter 2007

Announcements

- Deadline for early drop is extended to Saturday 11:59 PM

Type conversions

- Many languages permit conversions among types
- For example, casts in C/C++/Java are program constructs that convert a value of one type to a value of another type

```
double d = 3.14159;  
int i = (int) d;
```

- A cast can be treated by the compiler as a function call to a type-conversion function

```
double d = 3.14159;  
int i = int(d);
```

Type coercions

- Some conversions among types may be performed automatically
- Automatic type conversions are called *coercions*
- For example, Java will coerce one argument to a binary numeric operator to type double if the other argument is of type double:

```
int k=8; double d=0.5;  
double x = k * d;
```
- The rules for type coercion are part of the type system of a language and need to be built into the type-checking functionality of a compiler or interpreter

Type coercions in Onyx

- From **Integer** to **Decimal**
 - Comparison expressions
 $3 < 3.3, 3.0 < 3.3 \rightarrow \text{true true}$
 - Function calls that expect a **Decimal** argument
declare function f(\$x as onyx.types.Decimal) {\$x};
f(1)
- From **Number** to **String** in comparison expressions
 $3 < "4", "3.0" < "3.00", 3.0 < "3.00", "3.0" < 3.00$
true true false false

More on Type Coercions

- From `AnySimpleType` to `Sequence`: function calls

```
declare function f($x as onyx.types.Sequence) {first($x)};  
f(1), f ((4,5,6)) → 1 4
```

- But not the other way around

```
declare function f($x as onyx.types.Integer) {$x};  
let $a := (3,()) return f($a)
```

```
<DynamicError column="25" line="2">
```

```
<ErrorMessage>Function with prototype f(onyx.types.Sequence) not  
found</ErrorMessage>
```

```
<PossibleMatch>onyx.types.AnyTypef(onyx.types.Integer)</PossibleMatch>
```

```
</DynamicError>
```

- Everything else must be done with explicit type casting functions, `Integer`, `String`, etc.

Static vs. dynamic type checking

- Static type checking is useful: the compiler can find some errors without your having to run your code
- For example in Java, the compiler can tell you that there's something wrong with this function definition:

```
int f( int k , String s ) { return k * s; }
```

...namely that you're trying to multiply an `int` and a `String`, which makes no sense

- And also if you call that function `f` in a context inappropriate for an `int`, the compiler can tell you there's something wrong with that as well

Static vs. dynamic type checking

- Dynamic type checking is useful, but it delays detection of type errors until program execution
- In Onyx, we can define a function like this...
`declare function f($k,$s) { $k * $s };`
- Dynamic type checking can certainly find an error in the use of a function at *runtime* e.g.:
`f(3,"hello")`
- Static type checking is not obvious

Type inference

- In a language without type declarations for variables or functions, some static type checking may still be done
- The possible types of an expression can be determined to some extent by the use of the expression, even without type declarations
- This is called *type inference*
- Type inference is often applied to the problem of determining the argument types and return types of a function from looking at the function body

Type inference

- Consider the Onyx function definition:
`declare function f($k,$s) { $k * $s };`
- Even without type declarations, we can infer the following statically:
 - Because the `*` operator requires it, `$k` and `$s` must both be of type `Integer`
 - Similarly, the return type of `f` must be a of type `Integer`
- A static type checker *can* actually find an error in this use of the function: `f(3,"hello")`
- Type declarations can make the job easier
`declare function f($k as <INT>,$s as <INT>){ $k * $s };`

Type checking of expressions

- Define a “type” attribute for expression nodes
- The value of the type attribute is a type expression, or a special type error value, e.g. `type_error`
- Types of literal constants can be determined syntactically
- Types of variables can be determined from their declarations, if the language has declarations
- Types of other expressions can be determined from the types of operator operands and the type rules for the operators, i.e. type inferencing

Introducing names

- In different languages, different constructs are used to introduce names
- For example, introducing the name of a variable can be done with a declaration statement in C:

```
int num;
```

- ... or with a for-clause in Onyx:

```
for $num in (1,2,3)
```

- ... or just by using it for the first time in FORTRAN:

```
NUM = 3
```

Introducing names

- Names of functions are introduced by defining them in Java (this also introduces the formal parameter names):

```
int add (int x, int y) {  
    return x+y;  
}
```

- ... or by declaring them (writing a prototype) in C:

```
int add(int,int);
```

- Languages may also have constructs for introducing names of types, packages, modules, etc., etc.

Function prototype matching in Onyx

- We may overload, but not override, function definitions
- To resolve a function call, the number and types of arguments must match the function prototype definition
- If an exact argument type match isn't possible, attempt type promotion (i.e., `Integer` to `Decimal`), otherwise, a static error is raised

```
declare function f ( $a as onyx.types.Decimal) {(2,$a)};
```

```
f(3) → 2 3
```

- Notice what happens with `anyType`

```
declare function f ( $a as onyx.types.Decimal) { (2,$a) };
```

```
declare function f ( $a as onyx.types.Integer) { (3,$a) };
```

```
declare function f ( $a ) { (4,$a) };
```

```
(f(3), f(3.0)) → 3 3 2 3.0
```

What about anyType?

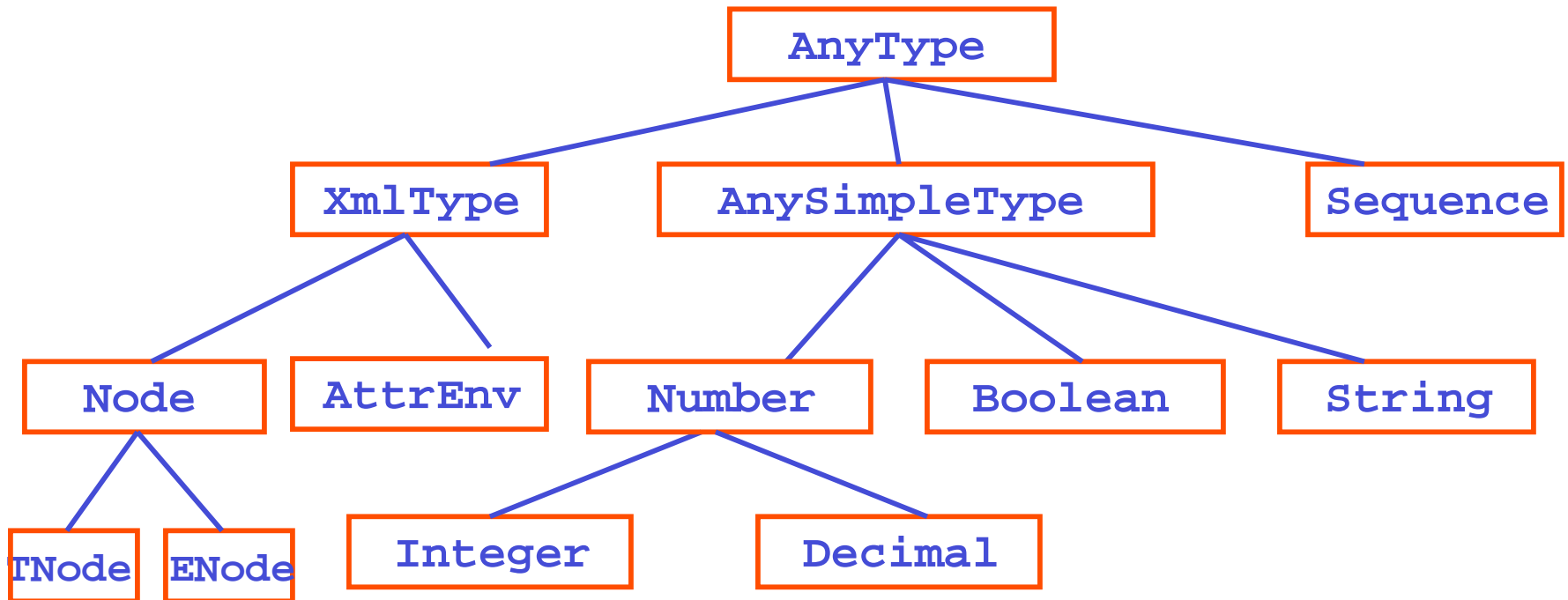
- Consider another example

```
declare function f($x) { "A" };  
declare function f($x as onyx.types.Decimal) { "B" };
```

f(9) → A

- Which type does `Integer` of an “isA” relationship to?

Type hierarchy in Onyx



Matches made in favor of IsA relationships

- Consider another example

```
declare function f($a) { (4,$a)};  
declare function f($a as onyx.types.Sequence) { (5,$a) };  
(f(3), f(3.0)) → 4 3 4 3.0
```

- When there is none, an error is raised:

```
declare function f ( $a as onyx.types.Decimal ) { (4,$a) };  
declare function f ( $a as onyx.types.Sequence ) { (5,$a) };  
(f(3), f(3.0)) →
```

```
<onyx.error.SemanticError>  
  <DynamicError column="2" line="3">  
    <ErrorMessage>Function with prototype f(onyx.types.Integer) not found  
    </ErrorMessage>  
    <PossibleMatch>onyx.types.AnyType f(onyx.types.Decimal)</PossibleMatch>  
    <PossibleMatch>onyx.types.AnyType f(onyx.types.Sequence)</PossibleMatch>  
  </DynamicError>  
</onyx.error.SemanticError>
```

Matching with multiple arguments

- If there are multiple matches, then the prototype requiring the least number of type promotions is used (# of arguments must agree)

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Decimal, $c as  
    onyx.types.Decimal) {"IDD"};
```

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Integer, $c as  
    onyx.types.Decimal) {"IID"};
```

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Integer, $c as  
    onyx.types.Integer) {"III"};
```

```
(f(3,4,5), f(3,4,5.0), f(3,4.0,5.0)) → III IID IDD
```

Multiple arguments

- If there are multiple matches, then the prototype requiring the least number of type promotions is used (# of arguments must agree)

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Decimal, $c as  
    onyx.types.Decimal) {"IDD"};
```

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Integer, $c as  
    onyx.types.Decimal) {"IID"};
```

f(3,4,5) → IID

Ambiguity with multiple matches

- If no function table entry matches the called function, or there is more than one entry requiring the least number of promotions, then a dynamic error is raised and all possible prototypes are displayed to the user

```
declare function f ( $a as onyx.types.Integer, $b as  
    onyx.types.Decimal, $c as onyx.types.Decimal) {"IDD"};
```

```
declare function f ( $a as onyx.types.Integer, $b as onyx.types.Integer,  
    $c as onyx.types.Decimal) {"IID"};
```

```
declare function f ( $a as onyx.types.Integer, $b as  
    onyx.types.Decimal, $c as onyx.types.Integer) {"IDI"};
```

f(3,4,5) → ?

Ambiguous matching

```
<onyx.error.SemanticError>  
  <DynamicError column="1" line="4">  
    <ErrorMessage>Function with prototype  
      f(onyx.types.Integer,onyx.types.Integer,onyx.types.Integer) not  
      found</ErrorMessage>  
    <PossibleMatch>onyx.types.AnyType  
      f(onyx.types.Integer,onyx.types.Decimal,onyx.types.Decimal)</Po  
      ssibleMatch>  
    <PossibleMatch>onyx.types.AnyType  
      f(onyx.types.Integer,onyx.types.Decimal,onyx.types.Integer)</Pos  
      sibleMatch>  
    <PossibleMatch>onyx.types.AnyType  
      f(onyx.types.Integer,onyx.types.Integer,onyx.types.Decimal)</Pos  
      sibleMatch>  
  </DynamicError>  
</onyx.error.SemanticError>
```

Error with Builtins

true + 1

```
<onyx.error.SemanticError>  
  <DynamicError column="6" line="1">  
    <ErrorMessage>Function with prototype  
      op:numeric-add(onyx.types.Boolean,onyx.types.Integer) not found  
    </ErrorMessage>  
    <PossibleMatch>onyx.types.Integer  
      op:numeric-add(onyx.types.Integer,onyx.types.Integer)  
    </PossibleMatch>  
    <PossibleMatch>onyx.types.String  
      op:numeric-add(onyx.types.String,onyx.types.String)  
    </PossibleMatch>  
  </DynamicError>  
</onyx.error.SemanticError>
```

Error with Builtins

"3" + 3

```
<onyx.error.SemanticError>
  <DynamicError column="5" line="1">
    <ErrorMessage>Function with prototype
      op:numeric-add(onyx.types.String,onyx.types.Integer) not found
    </ErrorMessage>
    <PossibleMatch>onyx.types.Integer
      op:numeric-add(onyx.types.Integer,onyx.types.Integer)
    </PossibleMatch>
    <PossibleMatch>onyx.types.String
      op:numeric-add(onyx.types.String,onyx.types.String)
    </PossibleMatch>
  </DynamicError>
</onyx.error.SemanticError>
```

Global variables, slight return

- The order of global variable definitions is significant, and forward references are not allowed
- In any context, a local binding obscures any global definitions

Examples with global variables

- See `~/../public/Materials/A4/Tests/varTests`

- Forward references are not allowed

```
declare variable $b { $a + 4 };
```

```
declare variable $a { 7 };
```

```
$b →
```

```
<DynamicError column="23" line="1">
```

```
  Variable $a not bound
```

```
</DynamicError>
```

Examples with global variables

- The order of a function definition with respect to a variable definition is not significant

```
declare variable $a { 7 };
```

```
declare function f() { $a + $b};
```

```
declare variable $b { 4 };
```

```
f() → 11
```

Examples with global variables

- Local bindings obscure global definitions

```
declare variable $a { let $a := 7 return $a };
```

```
declare variable $b as onyx.types.Integer {$a + 4};
```

```
let $a := $a + $b
```

```
return ($a,$b)
```

→ 18 11

Syntax Directed Translation

Syntax and semantics

- The syntax of an input language is defined with context-free grammar rules
- But translation involves more than meeting a context-free definition
- What remains are the semantic aspects of the input
- How do we connect the semantic rules with the syntactic rules of the grammar?
- We'll consider *syntax directed definitions*

Semantic rules and grammar symbol attributes

- The parser traces out a parse tree: nodes in the parse tree correspond to symbols in the grammar
- Grammar symbols and parse tree nodes can have *attributes* associated with them
- Think of a parse tree node as an object or record: an attribute is an instance variable of the node
- *Semantic rules* associated with grammar rules can compute the values of attributes and perform other actions

Syntax directed definitions

- A syntax directed definition is a generalization of a context free grammar
- Each grammar symbol has associated attributes
- The attributes can be anything: string, number, type, address, AST node, etc.
- Each rule of the grammar has a semantic rule associated with it, specifying how to compute attributes for symbols that appear in that rule
- We form the associations by labeling (“decorating”) nodes of the parse tree with attributes: we obtain an *annotated parse tree*
- The process is called *annotation* or *decoration*

Synthesized and inherited attributes

- Grammar rules are of the form
 $A ::= X_1 X_2 \dots X_N$, where A is a nonterminal and the X_i are terminal or nonterminal
- In a part of a parse tree corresponding to this rule, A corresponds to a parent node; the X_i correspond to siblings that are all children of that parent
- A semantic rule associated with this grammar rule can specify the value of an attribute of any of the symbols in the rule in a way that depends on the attributes of the others

Synthesized and inherited attributes

- A semantic rule that computes an attribute for A , depending on attributes of the X_i , is computing a *synthesized* attribute: it depends only on the attributes of its children
- A semantic rule that computes an attribute for one of the X_i , depending on attributes of A and the other X_j , is computing an *inherited* attribute: it depends on the attributes of its parent and siblings

Synthesized vs. inherited attributes

- Let $A \rightarrow X_1 X_2 \dots X_N$ be a production
- Let the semantic rule associated with the production be $b := f(c_1, c_2, \dots, c_k)$, where f is a function
- b and the c_i are attributes of the nodes for the symbols in the production
- We say that b is a *synthesized attribute* if it is an attribute of A
- ... and c_1, c_2, \dots, c_k are attributes associated with grammar symbols X_1, X_2, \dots, X_N (RHS of the production)
- We say that b is an *inherited attribute* if it is an attribute of one of the X_1, X_2, \dots, X_N (RHS of the production)...
- ...and c_1, c_2, \dots, c_k are attributes belonging to the other grammar symbols of the production A, X_1, X_2, \dots, X_N (LHS or RHS)

S-attributed definitions

- It is common to use only synthesized attributes
- Synthesized attributes can be computed naturally in a bottom-up parser
 - When a production is reduced, the attributes c_1, c_2, \dots, c_k of the RHS symbols have already been computed
 - The attribute of the LHS symbol can then be computed as
$$b := f(c_1, c_2, \dots, c_k)$$
- A set of semantic rules that uses only synthesized attributes is using *S-attributed definitions*

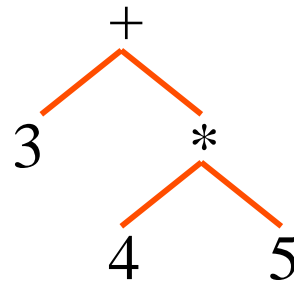
S-attributed definitions and syntax trees

- A typical use of S-attributed definitions is to build an AST
- Attributes are AST nodes and their instance variables
- An AST is better suited to the translation task than an exact copy of the parse tree

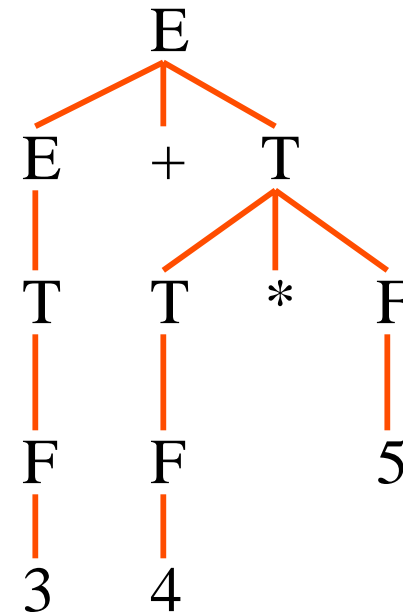
Parse trees and abstract syntax trees

$E ::= E + T \mid T$
 $T ::= T * F \mid F$
 $F ::= \text{digit}$

input: 3 + 4 * 5



Abstract Syntax Tree



Parse Tree

Using an AST

- With a successful parse, a suitable AST is built
- Now a relatively simple traversal of the AST can produce the desired result, e.g.
 - Perform a postorder traversal to translate to postfix notation: $3\ 4\ 5\ *\ +$
 - Perform actions associated with the operators at internal nodes to compute the result:
 $(3+(4*5)) = 23$

Single vs. multiple pass translation

- Building an AST that persists after the parse is completed, and then traversing the AST to perform the translation, is a multiple-pass approach
- This may be more space and time intensive than necessary; sometimes a single-pass translation approach will suffice
- Instead of building an AST, semantic definitions compute attribute values and perform the desired translation directly
- Let's look at 2 examples

Example: a desk calculator

- Each of the non-terminals E , T , and F have an associated integer-valued attribute val
- There is a semantic rule for each production
- It computes the val attribute for the non-terminal on the LHS of the production using the values for the non-terminals on the RHS

Production	Semantic rules
$L \rightarrow E ;$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathbf{digit}$	$F.val := \mathbf{digit.lexval}$

Example: a type declaration

- A declaration, generated by the non-terminal D
- Keywords are **int** or **real**, followed by a list of identifiers, e.g.: `int a, b, c`
- Non-terminal L has an inherited attribute in
- `addtype()` is a function that updates a symbol table with type information

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

Dependencies in attribute definition

- Synthesized attributes can be computed whenever all attributes for symbols on a rule's RHS are known
 - This is easy for a LR parser, harder for a LL parser
- Inherited attributes can require knowing attributes for symbols on a rule's LHS
 - This is easy for a LL parser, harder for a LR parser
- In general an attribute can be computed only when all its inputs are available
- With a mix of synthesized and inherited attributes, it can be tricky to make sure attributes are computed in the right order

Dependency graphs and topological sorts

- Dependencies among attribute computations form a dependency graph
- One approach:
 - Construct the attribute dependency graph at parse time
 - Compute a topological sort of the dependency graph (this preserves dependency constraints), and evaluate semantic rules accordingly
- This is a multiple-pass approach however, which we might want to avoid

Depth-first traversals and L-attributed definitions

- A very common (but not completely general) situation is to perform a pre-order traversal of the resulting parse tree, carrying out semantic actions
- We require that the inherited attributes of an AST node depend only on the inherited attributes of the siblings to the left (predecessors)
- A semantic rule with this property is called an *L-attributed definition*
- Called “L-attributed” because the attribute information appears to flow left-to-right

L-attributed and S-attributed definitions

- An L-attributed definition permits a mix of synthesized and inherited attributes
- An S-attributed definition is a special case: it has only synthesized attributes
- L-attributed definitions are more powerful
- Both LL and LR parsers are applicable

SDT approach to an interpreter

- The parser takes an input program and while parsing...
- ... either directly does the computations needed according to the semantics of the input program,
- ... or emits code which when interpreted carries out the computations needed according to the semantics of the input program

Tradeoffs

- The AST approach requires multi-pass translation
- The separation of AST traversal from parser actions makes for a cleaner design
- Some operations become easier when we have an explicit AST, such as optimization transformations
- Time and space costs associated with multi-pass techniques
- However, memory and CPU cycles are cheap

Static checks

- Recall that static checks of program correctness can be performed without running the program
- All syntactic checks
(unless we have run time code generation)
- Some semantic checks, e.g. correctly formed expressions

`1 + true`

- All other semantic checks must be handled dynamically while the program runs
 - Division by zero
 - File not found

Other examples of static checks

- Matching XML tag names: the QNames in an open tag and its matching close tag must be the same `<a> `
not `<a> ... `
- Uniqueness checks: case labels in a switch statement must not be duplicated
- Scope checks: a variable must not be used before it is declared
- Type checks: the types of expressions used as arguments to functions or operators must be compatible with the requirements
- Type checking is crucial, as it tells the compiler which version of an overloaded function to call

Type systems

- A *type system* is a collection of rules for assigning type expressions to various parts of a program
- A *type expression* is an expression that refers to a type
float, pointer to float, array of float, etc.
- A *type checker* implements a type system by annotating a program type expressions and testing if these annotations are permitted
- We can do this while traversing the AST
- Or we can do this in a syntax directed manner

Type checking of expressions

- Define a “type” attribute for expression nodes
- The value of the type attribute is a type expression, or a special type error value, e.g. `type_error`
- Types of literal constants can be determined syntactically
- Types of variables can be determined from their declarations, if the language has declarations
- Types of other expressions can be determined from the types of operator operands and the type rules for the operators, i.e. type inferencing

An example of type checking

- Example S-attributed definitions for type checking in a Pascal-like language
- The mod operator can take only int arguments

```
E ::= charLiteral    { E.type = char; }
E ::= intLiteral     { E.type = int; }
E ::= var             { E.type = Table.lookup(var.val); }
E ::= E1 mod E2      { E.type =
                       (E1.type = int and E2.type = int ) ? int :
                       type_error; }
```