Beyond Syntax
There is a level of correctness that is deeper than grammar

```c
fie(a,b,c,d)
    int a, b, c, d;
    { ... }

fee() {
    int f[3], g[0],
        h, i, j, k;
    char *p;
    fie(h,i,"ab",j, k);
    k = f * i + j;
    h = g[17];
    printf("<%s,%s>\n", p,q);
    p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

• declared g[0], used g[17]
• wrong number of args to fie()
• “ab” is not an int
• wrong dimension on use of f
• undeclared variable q
• 10 is not a character string

All of these are
“deeper than syntax”

To generate code, we need to understand its meaning!
Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is “x” a scalar, an array, or a function? Is “x” declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of “x” does each use reference?
- Is the expression “x * y + z” type-consistent?
- In “a[i,j,k]”, does a have three dimensions?
- Where can “z” be stored? (register, local, global, heap, static)
- In “f ← 15”, how should 15 be represented?
- How many arguments does “fie()” take? What about “printf ()”?
- Does “*p” reference the result of a “malloc()”?
- Do “p” & “q” refer to the same memory location?
- Is “x” defined before it is used?

These are beyond a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars?
  - Syntax directed translation using attribute grammars? (attributed grammars?)
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code (action routines)

In scanning & parsing, formalism won; different story here.
Beyond Syntax

Telling the story using Syntax Directed Definitions with attribute grammars:

• The attribute grammar formalism is important
  > Succinctly makes many points clear
  > Sets the stage for actual, \textit{ad-hoc} practice

• The problems with attribute grammars motivate practice
  > Non-local computation
  > Need for centralized information

• Some folks in the community still argue for attribute grammars
  > Knowledge is power
  > Information is immunization

We will cover attribute grammars, then move on to \textit{ad-hoc} ideas

Attribute grammars

• Generalization of context-free grammar
• Each grammar symbol has an associated set of attributes
• Augment grammar with rules that define values
• High-level specification, independent of evaluation scheme

Dependences between attributes

• Values are computed from constants & other attributes
• Synthesized attribute - value computed from children
• Inherited attribute - value computed from siblings & parent
**Syntax Directed Definitions (SDD)**

What is an attribute grammar?

- A context-free grammar augmented with a set of rules
- Each symbol in the derivation has a set of values, or *attributes*
- The rules specify how to compute a value for each attribute

**Example grammar**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>→ Sign List</td>
<td>This grammar describes signed binary numbers</td>
</tr>
<tr>
<td>Sign</td>
<td>→ +</td>
<td>We would like to augment it with rules that compute the decimal value of each valid input string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>→ List Bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ Bit</td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>→ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ 1</td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

*For “–1”*

Number → Sign List
Sign → – List
Sign → – Bit
Sign → – 1

*For “–101”*

Number → Sign List
Sign → Sign List Bit
Sign → Sign List 1
Sign → Sign List 1 1
Sign → Sign List 0 1
Sign → Sign 1 0 1
Sign → – 101

We will use these two throughout the lecture
Attribute Grammars

Add rules to compute the decimal value of a signed binary number

<table>
<thead>
<tr>
<th>Productions</th>
<th>Attribution Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number → Sign List</td>
<td>List.pos ← 0</td>
</tr>
<tr>
<td>Number.val ← If Sign.neg then - List.val else List.val</td>
<td></td>
</tr>
<tr>
<td>Sign → +</td>
<td>Sign.neg ← false</td>
</tr>
<tr>
<td></td>
<td>Sign.neg ← true</td>
</tr>
<tr>
<td>List_0 → List, Bit</td>
<td>List_1.pos ← List_0.pos + 1</td>
</tr>
<tr>
<td>Bit.pos ← List_0.pos</td>
<td></td>
</tr>
<tr>
<td>Bit.val ← List_0.val + Bit.val</td>
<td></td>
</tr>
<tr>
<td>List → Bit</td>
<td>Bit.pos ← List.pos</td>
</tr>
<tr>
<td>List.val ← Bit.val</td>
<td></td>
</tr>
<tr>
<td>Bit → 0</td>
<td>Bit.val ← 0</td>
</tr>
<tr>
<td>Bit → 1</td>
<td>Bit.val ← 2^Bit.pos</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>val</td>
</tr>
<tr>
<td>Sign</td>
<td>neg</td>
</tr>
<tr>
<td>List</td>
<td>pos, val</td>
</tr>
<tr>
<td>Bit</td>
<td>pos, val</td>
</tr>
</tbody>
</table>

Back to the Examples

For “-1”

One possible evaluation order:

1. List.pos
2. Sign.neg
3. Bit.pos
4. Bit.val
5. List.val
6. Number.val

Other orders are possible

Knuth suggested a data-flow model for evaluation

- Independent attributes first
- Others in order as input values become available

Evaluation order must be consistent with the attribute dependence graph

Rules + parse tree imply an attribute dependence graph
This is the complete attribute dependence graph for “–101”.
It shows the flow of all attribute values in the example.
Some flow downward
→ inherited attributes
Some flow upward
→ synthesized attributes
A rule may use attributes in the parent, children, or siblings of a node.

**The Rules of the Game**

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Label identical terms in production for uniqueness
- Rules & parse tree define an attribute dependence graph
  > Graph must be non-circular
This produces a high-level, functional specification

**Synthesized attribute**
> Depends on values from children

**Inherited attribute**
> Depends on values from siblings & parent
**Using Attribute Grammars**

Attribute grammars can specify context-sensitive actions
- Take values from syntax
- Perform computations with values
- Insert tests, logic, ...

<table>
<thead>
<tr>
<th>Synthesized Attributes</th>
<th>Inherited Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Use values from children &amp; from constants</td>
<td>- Use values from parent, constants, &amp; siblings</td>
</tr>
<tr>
<td>- S-attributed grammars</td>
<td>- directly express context</td>
</tr>
<tr>
<td>- Evaluate in a single bottom-up pass</td>
<td>- can rewrite to avoid them</td>
</tr>
<tr>
<td>Good match to LR parsing</td>
<td>- Thought to be more natural</td>
</tr>
<tr>
<td></td>
<td>- Not easily done at parse time</td>
</tr>
</tbody>
</table>

*We want to use both kinds of attribute*

---

**Evaluation Methods**

Dynamic, dependence-based methods
- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Define attributes in topological order

Rule-based methods
- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

Oblivious methods
- Ignore rules & parse tree
- Pick a convenient order (at design time) & use it

*Imagine a diminutive person who knows how to evaluate attributed trees ...*
Back to the Example

For “–101”

Back to the Example

For “–101”
Back to the Example

For 

Inherited Attributes

Synthesized attributes

Back to the Example
Back to the Example

Synthesized attributes

For “–101”

If we show the computation …

& then peel away the parse tree …
Back to the Example

All that is left is the attribute dependence graph. This succinctly represents the flow of values in the problem instance. The dynamic methods sort this graph to find independent values, then work along graph edges. The rule-based methods try to discover “good” orders by analyzing the rules. The oblivious methods ignore the structure of this graph.

The dependence graph must be acyclic

Circularity

We can only evaluate acyclic instances

• We can prove that some grammars can only generate instances with acyclic dependence graphs
• Largest such class is “strongly non-circular” grammars (SNC)
• SNC grammars can be tested in polynomial time
• Failing the SNC test is not conclusive

Many evaluation methods discover circularity dynamically
⇒ Bad property for a compiler to have

SNC grammars were first defined by Kennedy & Warren
A Circular Attribute Grammar

<table>
<thead>
<tr>
<th>Productions</th>
<th>Attribution Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong> → <strong>List</strong></td>
<td>Lists.a ← 0</td>
</tr>
<tr>
<td><strong>List</strong>₀ → <strong>List, Bit</strong></td>
<td>Lists.a ← Lists.a + 1</td>
</tr>
<tr>
<td></td>
<td>Lists.b ← Lists.b</td>
</tr>
<tr>
<td></td>
<td>Lists.c ← Lists.b + Bit.val</td>
</tr>
<tr>
<td></td>
<td>Bit</td>
</tr>
<tr>
<td><strong>Bit</strong> → 0</td>
<td>Bit.val ← 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

An Extended Example

Grammar for a basic block (§ 4.3.3)

Block₀ → Block₁ Assign
Assign → Ident = Expr ;
Expr₀ → Expr₁ + Term
| Expr₁ – Term |
| Term |
Term₀ → Term₁ * Factor
| Term₁ / Factor |
| Factor |
Factor → ( Expr )
| Number |
| Identifier |

Let’s estimate cycle counts
• Each operation has a COST
• Add them, bottom up
• Assume a load per value
• Assume no reuse
Simple problem for an AG

Hey, this looks useful!
Adding attribution rules

| Block₀ → Block₁ Assign | Block₀.cost ← Block₁.cost + Assign.cost |
| Block₀.cost ← Assign.cost |
| Assign → Ident = Expr ; | Assign.cost ← COST(store) + Expr.cost |
| Expr₀ → Expr₁ + Term | Expr₁.cost ← Expr₁.cost + COST(add) + Term.cost |
| Expr₁.cost ← Expr₁.cost + COST(add) + Term.cost |
| | Expr₁.cost ← Expr₁.cost + COST(add) + Term.cost |
| | Expr₁.cost ← Expr₁.cost + COST(add) + Term.cost |
| | Expr₁.cost ← Expr₁.cost + COST(add) + Term.cost |
| Term → Term₁ * Factor | Term₁.cost ← Term₁.cost + COST(mult) + Factor.cost |
| Term₁.cost ← Term₁.cost + COST(mult) + Factor.cost |
| | Term₁.cost ← Term₁.cost + COST(div) + Factor.cost |
| | Term₁.cost ← Term₁.cost + COST(div) + Factor.cost |
| Factor → ( Expr ) | Factor.cost ← Expr.cost |
| Factor.cost ← Expr.cost |
| | Number Factor.cost ← COST(loadi) |
| | Identifier Factor.cost ← COST(load) |
| | Identifier Factor.cost ← COST(load) |

All the attributes are synthesized!

An Extended Example (continued)

Properties of the example grammar

- All attributes are synthesized ⇒ S-attributed grammar
- Rules can be evaluated bottom-up in a single pass
  - Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement?

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded
Adding load tracking

- Need sets Before and After for each production
- Must be initialized, updated, and passed around the tree

<table>
<thead>
<tr>
<th>Factor</th>
<th>( Expr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Factor. cost ← Expr.cost ;</td>
</tr>
<tr>
<td>Identifier</td>
<td>Expr.Before ← Factor.Before ;</td>
</tr>
<tr>
<td></td>
<td>Factor.After ← Expr.After</td>
</tr>
<tr>
<td></td>
<td>Factor. cost ← COST(load) ;</td>
</tr>
<tr>
<td></td>
<td>Factor.After ← Factor.Before</td>
</tr>
<tr>
<td></td>
<td>If (Identifier.name ∉ Factor.Before) then</td>
</tr>
<tr>
<td></td>
<td>Factor. cost ← COST(load) ;</td>
</tr>
<tr>
<td></td>
<td>Factor.After ← Factor.Before ∪ Identifier.name</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>Factor. cost ← 0</td>
</tr>
<tr>
<td></td>
<td>Factor.After ← Factor.Before</td>
</tr>
</tbody>
</table>

Load tracking adds complexity
- But, most of it is in the “copy rules”
- Every production needs rules to copy Before & After

A sample production

<table>
<thead>
<tr>
<th>Expr₀</th>
<th>Expr₁ + Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr₀.cost ← Expr₁.cost + COST(add) + Term.cost ;</td>
<td></td>
</tr>
<tr>
<td>Expr₁.Before ← Expr₀.Before ;</td>
<td></td>
</tr>
<tr>
<td>Term.Before ← Expr₁.After ;</td>
<td></td>
</tr>
<tr>
<td>Expr₀.After ← Term.After</td>
<td></td>
</tr>
</tbody>
</table>

These copy rules multiply rapidly
- Each creates an instance of the set
- Lots of work, lots of space, lots of rules to write
What about accounting for finite register sets?

- **Before & After** must be of limited size
- Adds complexity to `Factor → Identifier`
- Requires more complex initialization

Jump from tracking loads to tracking registers is small

- Copy rules are already in place
- Some local code to perform the allocation

Next

- Curing these problems with *ad-hoc* syntax-directed translation

---

### An Even Better Model

Remember the previous example?

Grammar for a basic block

- `Block_0 → Block_1 Assign`  
  - `Assign`  
  - `Ident = Expr ;`  
  - `Expr_1 + Term`  
  - `Term_1 – Term`  
  - `Term_0`  
  - `Term_1 * Factor`  
  - `Term_1 / Factor`  
  - `( Expr )`  
  - `Number`  
  - `Identifier`

Let’s estimate cycle counts

- Each operation has a COST
- Add them, bottom up
- Assume a load per value
- Assume no reuse

Simple problem for an AG

Hey, this looks useful!
And Its Extensions

Tracking loads
- Introduced Before and After sets to record loads
- Added $\geq 2$ copy rules per production
  - Serialized evaluation into execution order
- Made the whole attribute grammar large & cumbersome

Finite register set
- Complicated one production ($\text{Factor} \rightarrow \text{Identifier}$)
- Needed a little fancier initialization
- Changes were quite limited

Why is one change hard and the other easy?

The Moral of the Story

- Non-local computation needed lots of supporting rules
- Complex local computation was relatively easy

The Problems
- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - Need copies of attributes
  - Can use pointers, for even more cognitive overhead
- Result is an attributed tree (somewhat subtle points)
  - Must build the parse tree
  - Must search tree for answers
**Addressing the Problem**

If you gave this problem to a chief programmer

- Introduce a central repository for facts
- Table of names
  - Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
  - Clean, efficient implementation
  - Good techniques for implementing the table *(hashing)*
  - When its done, information is in the table! *(no navigation)*
  - Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - Do we care?

**The Realist’s Alternative**

*Ad-hoc* syntax-directed translation

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
  - Includes ability to do tasteless & bad things

**To make this work**

- Need names for attributes of each symbol on *lhs & rhs*
  - Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, ...)
  - Yacc introduced $$, $1, $2, ..., $n, left to right
    
    \[ A ::= B C (\$$ = \text{concat}($1,$2)) \]

- Need an evaluation scheme
  - Fits nicely into LR(1) parsing algorithm
Reworking the Example
(with load tracking)

This looks cleaner & simpler!

Example — Building a Parse Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc syntax

Goal $\rightarrow$ Expr
  $\rightarrow$ Expr + Term
  $\rightarrow$ Expr − Term
  $\rightarrow$ Term
  $\rightarrow$ ( Expr )
  $\rightarrow$ number
  $\rightarrow$ id

Expr $\rightarrow$ Term
  $\rightarrow$ Factor

Term $\rightarrow$ Factor

Factor

Goal $\rightarrow$ Expr
  $\rightarrow$ Expr + Term
  $\rightarrow$ Expr − Term
  $\rightarrow$ Term
  $\rightarrow$ ( Expr )
  $\rightarrow$ number
  $\rightarrow$ id

$S = S + 1$

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;

$S = S + $1;
**Reality**

Most parsers are based on this *ad-hoc* style of context-sensitive analysis.

**Advantages**

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

**Disadvantages**

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc-like notation.

---

**Typical Uses**

- **Building a symbol table**
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses
    - *Note:* this assumes table is global

- **Simple error checking/type checking**
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformability of expression → bottom-up walk
  - Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Create list of sites to check
    - Either check offline, or handle the cases for arbitrary orderings
Is This Really “Ad-hoc”?  
Relationship between practice and attribute grammars

Similarities
• Both rules & actions associated with productions
• Application order determined by tools, not author
• (Somewhat) abstract names for symbols

Differences
• Actions applied as a unit; not true for AG rules
• Anything goes in ad-hoc actions; AG rules are functional
• AG rules are higher level than ad-hoc actions

Limitations
• Forced to evaluate in a given order: postorder
  > Left to right only
  > Bottom up only

• Implications
  > Declarations before uses
  > Context information cannot be passed down
    → How do you know what rule you are called from within?
    → Example: cannot pass bit position from right down

  \[ \text{List} \rightarrow \text{List Bit} : \] 
  \[ $$ = 2^*1 + 2 \]

  > Could you use globals?
    → In this case we could get the position from the left, which is not much help (and it requires initialization)
Alternative Strategy

- Build Abstract Syntax Tree
  - Use tree walk routines
  - Use “visitor” design pattern to add functionality

Visitor Treewalk I

- Parallel structure of tree:
  - Separates treewalk code from node handling code
  - Facilitates processing change without change to tree structure
**Visitor Treewalk II**

VisitAssignment(aNodePtr)

// preprocess assignment
(aNodePtr->rhs)->Accept(this);

// postprocess rhs info;
(aNodePtr->lhs)->Accept(this);

// postprocess assignment;

To start the process:

AnalysisVisitor a; treeRoot->Accept(a);

---

**Summary: Strategies for Context-Sensitive Analysis**

- **Attribute Grammars**
  - Pros: Formal, powerful, can deal with propagation strategies
  - Cons: Too many copy rules, no global tables, works on parse tree

- **Postorder Code Execution**
  - Pros: Simple and functional, can be specified in grammar (Yacc) but does not require parse tree
  - Cons: Rigid evaluation order, no context inheritance

- **Generalized Tree Walk**
  - Pros: Full power and generality, operates on abstract syntax tree (using Visitor pattern)
  - Cons: Requires specific code for each tree node type, more complicated
**Type systems**

**Types**
- Values that share a set of common properties defined by language and/or programmer

**Type system**
1. set of types in a programming language, and
2. rules that use types to specify program behavior

**Example type rules**
- If operands of addition are of type integer, then result is of type integer
- The result of the unary & operator is a pointer to the object referred to by the operand

**Advantages of typed languages**
- Ensure run-time safety
- Expressiveness (overloading, polymorphism)
- Provide information for code generation

---

**Type checking**

**Type checker**
- Enforces rules of type system
- May be strong/weak, static/dynamic

**Static type checking**
- Performed at compile time
- Early detection, no run-time overhead
- Not always possible (e.g., A[i])

**Dynamic type checking**
- Performed at run time
- More flexible, rapid prototyping
- Overhead to check run-time type tags
Type expressions

Type expressions
• Used to represent the type of a language construct
• Describes both language and programmer types

Examples
• Basic types: integer, real, character, ...
• Constructed types: arrays, records, pointers, functions,...

Constructing new types
• arrays \( array(1..10, T) \)
• records \( T_1 \times T_2 \ldots \)
• pointers \( pointer(T) \)
• functions \( T_1 \times T_2 \times \ldots \rightarrow T_n \)

A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:
\[
\begin{align*}
P & ::= L ; E \\
L & ::= L ; D | D \\
D & ::= D ; E | \text{id: } T \\
T & ::= \text{char} | \text{integer} | \text{array [num] of } T | \uparrow T \\
E & ::= \text{literal} | \text{num} | \text{id} | E \mod E | E[E] | E^{\uparrow}
\end{align*}
\]

Basic types \text{char, integer, typeError}

assume all arrays start at 1, e.g., \text{array [256] of char} results in the type expression \text{array(1..256, char)}

• \( \uparrow \) builds a pointer type, so \( \uparrow \text{integer} \) results in the type expression \text{pointer(integer)}
Type checking example

Partial attribute grammar for the type system

\[
L ::= L D \\
L ::= D \\
D ::= id: T  \quad \text{\{} \text{addtype(id.entry, T.type)} \text{\}} \\
T ::= \text{char} \quad \{T.type \leftarrow \text{char}\} \\
T ::= \text{integer} \quad \{T.type \leftarrow \text{integer}\} \\
T ::= \uparrow T_1 \quad \{T.type \leftarrow \text{pointer(T1.type)}\} \\
T ::= \text{array}[\text{num}]\text{ of }T_1 \quad \{T.type \leftarrow \text{array}(1:..\text{num.val}, T_1.type)\}
\]

Type checking expressions

Each expression is assigned a type using rules associated with the grammar.

- \( E ::= \text{literal} \quad \{E.type \leftarrow \text{char}\} \)
- \( E ::= \text{num} \quad \{E.type \leftarrow \text{integer}\} \)
- \( E ::= \text{id} \quad \{E.type \leftarrow \text{lookup(id.entry)}\} \)
- \( E ::= E_1 \mod E_2 \quad \{E.type \leftarrow \text{if }E_1.type = \text{integer and }E_2.type = \text{integer then integer else typeError}\} \)
- \( E ::= E_1[E2] \quad \{E.type \leftarrow \text{if }E_2.type = \text{integer and }E_1.type = \text{array}(s,t)\text{ then }t\text{ else }\text{typeError}\} \)
- \( E ::= E_1\uparrow \quad \{E.type \leftarrow \text{if }E_1.type = \text{pointer then }t\text{ such that pointer(E_1,t)}\text{ else }\text{typeError}\} \)
Type checking statements

Statements do not typically have values, therefore we assign them the type `void`. If an error is detected within the statement, it gets type `typeError`.

\[
S ::= id \leftarrow E \quad \{ S.type \leftarrow \text{if } id.type = E.type \text{ then void else typeError} \}
\]

\[
S ::= \text{if } E \text{ then } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean then } S_1.type \text{ else typeError} \}
\]

\[
S ::= \text{while } E \text{ do } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean then } S_1.type \text{ else typeError} \}
\]

\[
S ::= S_1 ; S_2 \quad \{ S.type \leftarrow \text{if } S_1.type = \text{void then } S_2.type \text{ else typeError} \}
\]

Type checking functions

We add two new productions to the grammar to represent function declarations and applications.

\[
T ::= T \rightarrow T \quad \text{declaration}
\]

\[
E ::= E ( E ) \quad \text{application}
\]

To capture the argument and return type, we use

\[
T ::= T_1 \rightarrow T_2 \quad \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \}
\]

\[
E ::= E_1 ( E_2 ) \quad \{ E.type \leftarrow \text{if } E_1.type = s \rightarrow t \text{ and } E_2.type = s \text{ then } t \text{ else typeError} \}
\]

Note: We could avoid functions `addtype` and `lookup` by adding a new inherited attribute `symboltable` which was a set that kept track of declarations, and was passed down parse tree. (How?)