Name Resolution and Type Checking

University of Waterloo

March 18, 2020
• What is the purpose of context-sensitive analysis?
What is the purpose of context-sensitive analysis?

Catch all remaining compile-time errors that were not caught by the scanner or parser.
• What is the purpose of context-sensitive analysis?
• Catch all remaining compile-time errors that were not caught by the scanner or parser.
• **Name errors**: duplicate declarations, use of undeclared or out-of-scope identifiers, etc.
Designing a Context-Sensitive Analyzer

- What is the purpose of context-sensitive analysis?
- Catch all remaining compile-time errors that were not caught by the scanner or parser.

- **Name errors**: duplicate declarations, use of undeclared or out-of-scope identifiers, etc.
- **Type errors**: returning the wrong type from a procedure, adding two things that can’t be added, etc.
- What is the purpose of context-sensitive analysis?
- Catch all remaining compile-time errors that were not caught by the scanner or parser.
- **Name errors**: duplicate declarations, use of undeclared or out-of-scope identifiers, etc.
- **Type errors**: returning the wrong type from a procedure, adding two things that can’t be added, etc.
- These are caught by **name resolution and type checking**.
• What is the purpose of context-sensitive analysis?
• Catch all remaining compile-time errors that were not caught by the scanner or parser.
• **Name errors**: duplicate declarations, use of undeclared or out-of-scope identifiers, etc.
• **Type errors**: returning the wrong type from a procedure, adding two things that can’t be added, etc.
• These are caught by **name resolution and type checking**.
• We build a **symbol table** mapping identifiers to types (or type signatures for procedures) to help with both phases.
In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.
The Parse Tree

- In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.
- In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.
The Parse Tree

- In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.
- In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.
- A .wlp4i file is a preorder traversal – constructing the tree is essentially the same as Assignment 3 Problem 1.
In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.

In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.

A .wlp4i file is a preorder traversal — constructing the tree is essentially the same as Assignment 3 Problem 1.

In A3P1 you were given the number of children for each node.
In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.

In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.

A .wlp4i file is a preorder traversal – constructing the tree is essentially the same as Assignment 3 Problem 1.

In A3P1 you were given the number of children for each node.

In a .wlp4i file you infer the number of children from the number of symbols on the right hand side of the rule.
• In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.

• In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.

• A .wlp4i file is a preorder traversal – constructing the tree is essentially the same as Assignment 3 Problem 1.

• In A3P1 you were given the number of children for each node.

• In a .wlp4i file you infer the number of children from the number of symbols on the right hand side of the rule.

• Terminal tokens correspond to nodes with no children.
In Assignment 7 you built a parse tree for the input and traversed it to produce a .wlp4i file.

In Assignment 8, the first thing you must do is reconstruct the tree from the .wlp4i file.

A .wlp4i file is a preorder traversal – constructing the tree is essentially the same as Assignment 3 Problem 1.

In A3P1 you were given the number of children for each node.

In a .wlp4i file you infer the number of children from the number of symbols on the right hand side of the rule.

Terminal tokens correspond to nodes with no children.

Rules with an empty right hand side also correspond to nodes with no children.
Suggested design for your parse tree class:

class Tree {
  private:
    string cachedType;
  public:
    string rule;
    string lhs, kind;
    string rhs, lexeme;
    vector<string> tokens;
    vector<Tree> children;
    Tree& child(string name, int occurrence = 1);
    string getType();
};

At each nonterminal node, store the rule used to expand the node.
The Parse Tree

- Suggested design for your parse tree class:

  ```
  class Tree {
    private:
      string cachedType;
    public:
      string rule;
      string lhs, kind;
      string rhs, lexeme;
      vector<string> tokens;
      vector<Tree> children;
      Tree& child(string name, int occurrence = 1);
      string getType();
  }
  ```

- Store the LHS and RHS of the rule. For terminal nodes, store the kind and lexeme.
The Parse Tree

• Suggested design for your parse tree class:

```cpp
class Tree {
    private:
        string cachedType;
    public:
        string rule;
        string lhs, kind;
        string rhs, lexeme;
        vector<string> tokens;
        vector<Tree> children;
        Tree& child(string name, int occurrence = 1);
        string getType();
}"

• Storing a tokenized copy of the rule may also be useful.
Suggested design for your parse tree class:

```cpp
class Tree {
    private:
        string cachedType;
    public:
        string rule;
        string lhs, kind;
        string rhs, lexeme;
        vector<string> tokens;
        vector<Tree> children;
        Tree& child(string name, int occurrence = 1);
        string getType();
}
```

- Store the sequence of children.
Suggested design for your parse tree class:

```cpp
class Tree {
    private:
        string cachedType;
    public:
        string rule;
        string lhs, kind;
        string rhs, lexeme;
        vector<string> tokens;
        vector<Tree> children;
        Tree& child(string name, int occurrence = 1);
        string getType();
}
```

Optional but very useful: write a function for looking up children by name. If there are multiple children with the same name, you can optionally pass in a number to indicate which you want.
The Parse Tree

- Suggested design for your parse tree class:
  ```cpp
class Tree {
    private:
      string cachedType;
    public:
      string rule;
      string lhs, kind;
      string rhs, lexeme;
      vector<string> tokens;
      vector<Tree> children;
      Tree& child(string name, int occurrence = 1);
      string getType();
  }
```

- A function for computing the type of a tree, if it has a type. Once you’ve computed the type, cache it and have the function return the cached type, so you don’t redo the type computation.
The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.
The Symbol Table

- The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.
- Types are represented by strings.
The Symbol Table

- The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.
- Types are represented by strings.
- Data structure: `map<string, pair<Signature, VariableTable>>` (procedure names to signatures and variable tables).
The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.

- Types are represented by strings.
- Data structure: `map<string, pair<Signature, VariableTable>>` (procedure names to signatures and variable tables).
- Signature is `vector<string>` (sequence of types).
The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.

Types are represented by strings.

Data structure: `map<string, pair<Signature, VariableTable>>` (procedure names to signatures and variable tables).

Signature is `vector<string>` (sequence of types).

VariableTable is `map<string, string>` (variable names to types).
The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.

Types are represented by strings.

Data structure: map<string, pair<Signature, VariableTable> (procedure names to signatures and variable tables).

Signature is vector<string> (sequence of types).

VariableTable is map<string, string> (variable names to types).

Final data structure is map<string, pair<vector<string>, map<string, string>>>.
The symbol table is a map that lets you look up the signature of a procedure, and the types of the local variables in the procedure.

- Types are represented by strings.
- Data structure: map<string, pair<Signature, VariableTable>> (procedure names to signatures and variable tables).
- Signature is vector<string> (sequence of types).
- VariableTable is map<string, string> (variable names to types).
- Final data structure is map<string, pair<vector<string>, map<string, string>>>.
- Wrap this data structure in a class with helper functions for working with it.
The Symbol Table

- Suggested design for your symbol table class:
  ```
  currentProcedure = "";
  ```
  ```
  typedef vector<string> Signature;
  typedef map<string, string> VariableTable;
  typedef map<string, pair<Signature, VariableTable>> ProcedureTable;
  ```
  ```
  class SymbolTable {
  private:
      ProcedureTable procedureTable;
  public:
      bool isAccessibleVariable(string name);
      bool isAccessibleProcedure(string name);
      void addVariable(Tree& tree);
      void addProcedure(Tree& tree);
      string lookupVariable(string name);
      Signature lookupProcedure(string name);
      void printProcedure(string name);
      void printTable();
  }
  ```

- Have a global variable that keeps track of the current procedure you are processing.
• Suggested design for your symbol table class:

```cpp
currentProcedure = ""

typedef vector<string> Signature;
typedef map<string, string> VariableTable;
typedef map<string, pair<Signature, VariableTable>> ProcedureTable;

class SymbolTable {
private:
    ProcedureTable procedureTable;
public:
    bool isAccessibleVariable(string name);
    bool isAccessibleProcedure(string name);
    void addVariable(Tree& tree);
    void addProcedure(Tree& tree);
    string lookupVariable(string name);
    Signature lookupProcedure(string name);
    void printProcedure(string name);
    void printTable();
}

• Have functions for checking if a variable or procedure is accessible in the scope of the current procedure.
• Suggested design for your symbol table class:

```cpp
std::string currentProcedure = "";

typedef std::vector<std::string> Signature;
typedef std::map<std::string, std::string> VariableTable;
typedef std::map<std::string, std::pair<Signature, VariableTable>> ProcedureTable;

class SymbolTable {
  private:
    ProcedureTable procedureTable;
  public:
    bool isAccessibleVariable(std::string name);
    bool isAccessibleProcedure(std::string name);
    void addVariable(Tree& tree);
    void addProcedure(Tree& tree);
    std::string lookupVariable(std::string name);
    Signature lookupProcedure(std::string name);
    void printProcedure(std::string name);
    void printTable();
};
```

• Have functions for adding a variable or procedure to the table, given a tree representing a variable or procedure declaration.
• Suggested design for your symbol table class:

```cpp
currentProcedure = "";

typedef vector<string> Signature;
typedef map<string, string> VariableTable;
typedef map<string, pair<Signature, VariableTable>> ProcedureTable;

class SymbolTable {
private:
    ProcedureTable procedureTable;
public:
    bool isAccessibleVariable(string name);
    bool isAccessibleProcedure(string name);
    void addVariable(Tree& tree);
    void addProcedure(Tree& tree);
    string lookupVariable(string name);
    Signature lookupProcedure(string name);
    void printProcedure(string name);
    void printTable();
}
```

• Have functions for looking up a variable or procedure and returning the type or type signature.
The Symbol Table

- Suggested design for your symbol table class:
  
  ```cpp
  currentProcedure = "";
  
  typedef vector<string> Signature;
  typedef map<string, string> VariableTable;
  typedef map<string, pair<Signature, VariableTable>> ProcedureTable;
  
  class SymbolTable {
    private:
      ProcedureTable procedureTable;
    public:
      bool isAccessibleVariable(string name);
      bool isAccessibleProcedure(string name);
      void addVariable(Tree& tree);
      void addProcedure(Tree& tree);
      string lookupVariable(string name);
      Signature lookupProcedure(string name);
      void printProcedure(string name);
      void printTable();
  }
  
  - Have functions for printing a particular procedure, or printing the whole table.
• `addVariable` should add the variable to the symbol table of the current procedure.
Implementation Notes

- `addVariable` should add the variable to the symbol table of the current procedure.
- `addProcedure` should update the current procedure.
• `addVariable` should add the variable to the symbol table of the current procedure.

• `addProcedure` should update the current procedure.

• `addVariable` and `addProcedure` should check if the variable or procedure is already in the table, and if so, produce a duplicate declaration error. Don’t use `isAccessibleProcedure` to check if a procedure is in the table – as we will see it requires some extra logic that makes using it for this check invalid in some cases.
Implementation Notes

- `addVariable` should add the variable to the symbol table of the current procedure.
- `addProcedure` should update the current procedure.
- `addVariable` and `addProcedure` should check if the variable or procedure is already in the table, and if so, produce a duplicate declaration error. Don’t use `isAccessibleProcedure` to check if a procedure is in the table – as we will see it requires some extra logic that makes using it for this check invalid in some cases.
- `addProcedure` will probably need separate cases for `wain` and for other procedures.
Implementation Notes

- `addVariable` should add the variable to the symbol table of the current procedure.
- `addProcedure` should update the current procedure.
- `addVariable` and `addProcedure` should check if the variable or procedure is already in the table, and if so, produce a duplicate declaration error. Don’t use `isAccessibleProcedure` to check if a procedure is in the table – as we will see it requires some extra logic that makes using it for this check invalid in some cases.
- `addProcedure` will probably need separate cases for `wain` and for other procedures.
- `lookupVariable` is relative to the current procedure.
Implementation Notes

- `addVariable` should add the variable to the symbol table of the current procedure.
- `addProcedure` should update the current procedure.
- `addVariable` and `addProcedure` should check if the variable or procedure is already in the table, and if so, produce a duplicate declaration error. Don’t use `isAccessibleProcedure` to check if a procedure is in the table – as we will see it requires some extra logic that makes using it for this check invalid in some cases.
- `addProcedure` will probably need separate cases for `wain` and for other procedures.
- `lookupVariable` is relative to the current procedure.
- `isAccessibleProcedure` is tricky to implement correctly – let’s see why on the next slide.
Implementation Notes

- `isAccessibleProcedure` is tricky to implement because variables can have the same name as the procedure containing them.

  ```
  // Valid program
  int sum(int* a, int n) {
    int sum = 0;
    while(n > 0) {
      n = n-1;
      sum = sum + *(a+n);
    }
    return sum;
  }

  int wain(int* a, int n) {
    return sum(a,n);
  }

  // Invalid program
  int sum(int* a, int n) {
    int sum = 0;
    if(n > 0) {
      sum = *a + sum(a+1,n-1);
    } else {}
    return sum;
  }

  int wain(int* a, int n) {
    return sum(a,n);
  }
  ```

- When a local variable has the same name as its enclosing procedure, all occurrences of this name refer to the variable rather than the procedure. This applies to parameters as well.

- Line `sum = *a + sum(a+1,n-1);` is invalid because `sum` refers to the variable, not the procedure.
isAccessibleProcedure is tricky to implement because variables can have the same name as the procedure containing them.

```c
// Valid program
int sum(int* a, int n) {
    int sum = 0;
    while(n > 0) {
        n = n-1;
        sum = sum + *(a+n);
    }
    return sum;
}

int wain(int* a, int n) {
    return sum(a,n);
}
```

```c
// Invalid program
int sum(int* a, int n) {
    int sum = 0;
    if(n > 0) {
        sum = *a + sum(a+1,n-1);
    } else {}
    return sum;
}

int wain(int* a, int n) {
    return sum(a,n);
}
```

- We implement isAccessibleProcedure(name) as follows:
  - Call isAccessibleVariable(name) to check if name is the name of a local variable in the current procedure. If it is, return false.
  - Otherwise simply check if the procedure is in the procedure table.
• To fill out the symbol table, traverse the parse tree and look at the rule at each node.
Filling out the Symbol Table

- To fill out the symbol table, traverse the parse tree and look at the rule at each node.
- Rules with `procedure` or `main` as the left hand side are procedure declarations. Pass the subtree into `addProcedure`. 
Filling out the Symbol Table

- To fill out the symbol table, traverse the parse tree and look at the rule at each node.
- Rules with `procedure` or `main` as the left hand side are procedure declarations. Pass the subtree into `addProcedure`.
- The rule `dcl type ID` is a variable declaration. Pass the subtree into `addVariable`.
To fill out the symbol table, traverse the parse tree and look at the rule at each node.

Rules with procedure or main as the left hand side are procedure declarations. Pass the subtree into addProcedure.

The rule dcl type ID is a variable declaration. Pass the subtree into addVariable.

Rules factor ID and lvalue ID are variable uses. Pass the lexeme of the ID into isAccessibleVariable. If it is not accessible, this is an error.
Filling out the Symbol Table

- To fill out the symbol table, traverse the parse tree and look at the rule at each node.
- Rules with procedure or main as the left hand side are procedure declarations. Pass the subtree into addProcedure.
- The rule dcl type ID is a variable declaration. Pass the subtree into addVariable.
- Rules factor ID and lvalue ID are variable uses. Pass the lexeme of the ID into isAccessibleVariable. If it is not accessible, this is an error.
- Rules factor ID LPAREN RPAREN and factor ID LPAREN arglist RPAREN are procedure calls. Pass the lexeme of the ID into isAccessibleProcedure. If it is not accessible, this is an error.
Filling out the Symbol Table

- To fill out the symbol table, traverse the parse tree and look at the rule at each node.
- Rules with procedure or main as the left hand side are procedure declarations. Pass the subtree into addProcedure.
- The rule dcl type ID is a variable declaration. Pass the subtree into addVariable.
- Rules factor ID and lvalue ID are variable uses. Pass the lexeme of the ID into isAccessibleVariable. If it is not accessible, this is an error.
- Rules factor ID LPAREN RPAREN and factor ID LPAREN arglist RPAREN are procedure calls. Pass the lexeme of the ID into isAccessibleProcedure. If it is not accessible, this is an error.
- You must do this in a single pass through the parse tree. If you do multiple passes, it will be difficult to catch errors involving things being used before they are declared.
Type checking has two parts:
Type Checking

- Type checking has two parts:
- Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
Type Checking

- Type checking has two parts:
  - Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
  - Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.
Type Checking

- Type checking has two parts:
  - Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
  - Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.
- Computing the types of subtrees with a type is done by implementing the `getType` function in the parse tree class.
Type checking has two parts:

Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.

Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.

Computing the types of subtrees with a type is done by implementing the `getType` function in the parse tree class.

Verifying well-typedness will require a custom tree traversal function, similar to building the symbol table.
Type Checking

- Type checking has two parts:
  - Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
  - Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.
- Computing the types of subtrees with a type is done by implementing the `getType` function in the parse tree class.
- Verifying well-typedness will require a custom tree traversal function, similar to building the symbol table.
- In the WLP4 specification, the following type rule divides the two parts of type checking:
Type Checking

- Type checking has two parts:
  - Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
  - Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.

- Computing the types of subtrees with a type is done by implementing the `getType` function in the parse tree class.

- Verifying well-typedness will require a custom tree traversal function, similar to building the symbol table.

- In the WLP4 specification, the following type rule divides the two parts of type checking:
  - “The second `dcl` in the sequence directly derived from `main` must derive a type that derives INT.”
Type Checking

- Type checking has two parts:
  - Part 1: Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
  - Part 2: Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.
- Computing the types of subtrees with a type is done by implementing the `getType` function in the parse tree class.
- Verifying well-typedness will require a custom tree traversal function, similar to building the symbol table.
- In the WLP4 specification, the following type rule divides the two parts of type checking:
  - “The second `dcl` in the sequence directly derived from `main` must derive a type that derives `INT`.”
- Everything before this rule is Part 1, everything after and including this rule is Part 2.
Here is the basic structure of the `getType` function:

```cpp
string getType() {
    if (cachedType != "") { return cachedType; }
    string type = "none";

    <main body>

    cachedType = type;
    return type;
}
```
Here is the basic structure of the `getType` function:

```c
string getType() {
    if(cachedType != "") { return cachedType; }
    string type = "none";

    <main body>
    cachedType = type;
    return type;
}
```

At the end of the function, we store the computed type in the tree’s `cachedType` field.
Here is the basic structure of the `getType` function:

```cpp
string getType() {
    if(cachedType != "") { return cachedType; }
    string type = "none";

    <main body>

    cachedType = type;
    return type;
}
```

At the end of the function, we store the computed type in the tree’s `cachedType` field.

At the start of the function, we return the cached type immediately if it has been computed.
Here is the basic structure of the `getType` function:

```cpp
string getType() {
    if(cachedType != "") { return cachedType; }
    string type = "none";

    <main body>

    cachedType = type;
    return type;
}
```

At the end of the function, we store the computed type in the tree’s `cachedType` field.

At the start of the function, we return the cached type immediately if it has been computed.

This way if we call `getType` twice on the same tree, we avoid redoing work.
Here is the basic structure of the `getType` function:

```cpp
string getType() {
    if(cachedType != "") { return cachedType; }
    string type = "none";

    <main body>

    cachedType = type;
    return type;
}
```

At the end of the function, we store the computed type in the tree’s `cachedType` field.

At the start of the function, we return the cached type immediately if it has been computed.

This way if we call `getType` twice on the same tree, we avoid redoing work.

Next let’s look at how to implement the main body, which actually computes the type.
The WLP4 specification gives a big list of type rules in English.
The WLP4 specification gives a big list of type rules in English. The main body should basically have a case for each of the Part 1 type rules.
The WLP4 specification gives a big list of type rules in English.
The main body should basically have a case for each of the Part 1 type rules.
For example:
The WLP4 specification gives a big list of type rules in English.

The main body should basically have a case for each of the Part 1 type rules.

For example:

- “The type of an ID is int if the dcl in which the ID is declared derives a sequence containing a type that derives INT.”
The WLP4 specification gives a big list of type rules in English.

The main body should basically have a case for each of the Part 1 type rules.

For example:

- “The type of an ID is \texttt{int} if the \texttt{dcl} in which the ID is declared derives a sequence containing a type that derives \texttt{INT}.”
- “The type of an ID is \texttt{int*} if the \texttt{dcl} in which the ID is declared derives a sequence containing a type that derives \texttt{INT STAR}.”
• The WLP4 specification gives a big list of type rules in English.
• The main body should basically have a case for each of the Part 1 type rules.
• For example:
  • “The type of an ID is int if the dcl in which the ID is declared derives a sequence containing a type that derives INT.”
  • “The type of an ID is int* if the dcl in which the ID is declared derives a sequence containing a type that derives INT STAR.”
• We would implement this by looking up the ID in our symbol table:
  ```java
  if(kind == "ID") {
    type = symbolTable.lookupVariable(lexeme);
  }
  ```
- The WLP4 specification gives a big list of type rules in English.
- The main body should basically have a case for each of the Part 1 type rules.
- For example:
  - “The type of an ID is int if the dcl in which the ID is declared derives a sequence containing a type that derives INT.”
  - “The type of an ID is int* if the dcl in which the ID is declared derives a sequence containing a type that derives INTSTAR.”
- We would implement this by looking up the ID in our symbol table:
  ```java
  if(kind == "ID") {
      type = symbolTable.lookupVariable(lexeme);
  }
  ```
- Recall that kind and lexeme are fields in our tree class, and type is the type that we cache and return.
Next let’s look at a nonterminal type rule:
Next let’s look at a nonterminal type rule:

“The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.”
Next let’s look at a nonterminal type rule:

- “The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.”

Here’s how we would add this rule to the main body:

```java
if(kind == "ID") {
    type = symbolTable.lookupVariable(lexeme);
} else if (rule == "factor AMP lvalue") {
    if(child("lvalue").getType() != "int") {
        output "ERROR: operator & expects int operand"
    }
    type = "int*";
}
```
Next let's look at a nonterminal type rule:

- “The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.”

Here's how we would add this rule to the main body:

```java
if(kind == "ID") {
    type = symbolTable.lookupVariable(lexeme);
} else if (rule == "factor AMP lvalue") {
    if(child("lvalue").getType() != "int") {
        output "ERROR: operator & expects int operand"
    }
    type = "int*";
}
```

We check for the production rule described in the type rule. Then we recursively call getType to find the type of the lvalue subtree.
Next let’s look at a nonterminal type rule:

- “The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.”

Here’s how we would add this rule to the main body:

```java
if(kind == "ID") {
    type = symbolTable.lookupVariable(lexeme);
} else if (rule == "factor AMP lvalue") {
    if(child("lvalue").getType() != "int") {
        output "ERROR: operator & expects int operand"
    }
    type = "int*";
}
```

We check for the production rule described in the type rule. Then we recursively call `getType` to find the type of the `lvalue` subtree.

This recursive call will fail right now because there are no cases for handling `lvalue` nodes. But...
• Next let’s look at a nonterminal type rule:
  • “The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.”
• Here’s how we would add this rule to the main body:
  ```java
  if(kind == "ID") {
    type = symbolTable.lookupVariable(lexeme);
  } else if (rule == "factor AMP lvalue") {
    if(child("lvalue").getType() != "int") {
      output "ERROR: operator & expects int operand"
    }
    type = "int*";
  }
  ```
• We check for the production rule described in the type rule. Then we recursively call `getType` to find the type of the `lvalue` subtree.
• This recursive call will fail right now because there are no cases for handling `lvalue` nodes. But...
• Once we implement all the Part 1 type rules here, the recursive call will compute the type of the `lvalue`, and we can use the computed type to determine if the `factor AMP lvalue` node is well-typed.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is factor ID LPAREN arglist RPAREN.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is factor ID LPAREN arglist RPAREN.
- Figure out the sequence of types specified by arglist, and compare it with the signature of the procedure specified by ID.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is `factor ID LPAREN arglist RPAREN`.
- Figure out the sequence of types specified by `arglist`, and compare it with the signature of the procedure specified by `ID`.
- The sequence of types must **exactly** match the signature.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is factor ID LPAREN arglist RPAREN.
- Figure out the sequence of types specified by arglist, and compare it with the signature of the procedure specified by ID.
- The sequence of types must exactly match the signature.
- The expr expr PLUS term and expr expr MINUS term rules are also a little complicated because they have several different cases to deal with.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is `factor ID LPAREN arglist RPAREN`.
- Figure out the sequence of types specified by `arglist`, and compare it with the signature of the procedure specified by `ID`.
- The sequence of types must exactly match the signature.
- The `expr expr PLUS term` and `expr expr MINUS term` rules are also a little complicated because they have several different cases to deal with.
- Make sure to call `getType` on every tree that has a defined type, or you might miss some type errors.
Once you get the general idea it’s just a matter of translating the English type rules into code. Some additional tips:

- Probably the most difficult rule to deal with is `factor ID LPAREN arglist RPAREN`.
- Figure out the sequence of types specified by `arglist`, and compare it with the signature of the procedure specified by `ID`.
- The sequence of types must exactly match the signature.
- The `expr expr PLUS term` and `expr expr MINUS term` rules are also a little complicated because they have several different cases to deal with.
- Make sure to call `getType` on every tree that has a defined type, or you might miss some type errors.
- Trees with the rule `dcl type ID` technically don’t have a defined type in the specification, you may want to make `getType` compute the type associated with the declaration anyways.
In this part, we traverse the parse tree and check that things like statements and declarations follow the type rules.
In this part, we traverse the parse tree and check that things like statements and declarations follow the type rules.

“The second dcl in the sequence directly derived from main must derive a type that derives int.”
In this part, we traverse the parse tree and check that things like statements and declarations follow the type rules.

“The second dcl in the sequence directly derived from main must derive a type that derives int.”

In your tree traversal you would have something like this:

```java
if(tree.lhs == "main") {
    if(tree.child("dcl",2).getType() != "int") {
        output "ERROR: second parameter of main should be int type"
    }
}
```
In this part, we traverse the parse tree and check that things like statements and declarations follow the type rules.

“The second dcl in the sequence directly derived from main must derive a type that derives int.”

In your tree traversal you would have something like this:

```java
if(tree.lhs == "main") {
    if(tree.child("dcl",2).getType() != "int") {
        output "ERROR: second parameter of main should be int type"
    }
}
```

“Whenever test directly derives a sequence containing two exprs, they must both have the same type.”
In this part, we traverse the parse tree and check that things like statements and declarations follow the type rules.

“The second dcl in the sequence directly derived from main must derive a type that derives int.”

In your tree traversal you would have something like this:

```java
if (tree.lhs == "main") {
    if (tree.child("dcl", 2).getType() != "int") {
        output "ERROR: second parameter of main should be int type"
    }
}
```

“Whenever test directly derives a sequence containing two exprs, they must both have the same type.”

Code:

```java
if (tree.lhs == "test") {
    leftType = tree.child("expr", 1).getType();
    rightType = tree.child("expr", 2).getType();
    if (leftType != rightType) {
        output "ERROR: mismatched types in conditional test"
    }
}
```
You need to make sure that every subtree that needs to be type-checked is actually visited.
• You need to make sure that every subtree that needs to be type-checked is actually visited.

• Easy way: just visit every node (but this is inefficient).
• You need to make sure that every subtree that needs to be type-checked is actually visited.
• Easy way: just visit every node (but this is inefficient).
• Faster but harder way: explicitly recurse on the subtrees that need to be checked.
You need to make sure that every subtree that needs to be type-checked is actually visited.

Easy way: just visit every node (but this is inefficient).

Faster but harder way: explicitly recurse on the subtrees that need to be checked.

For example, for the main rule:

```java
if(tree.lhs == "main") {
    // check the second dcl and the return expr
    // now recurse on the parts that need checking: the dcls and statements
    typeChecking(tree.child("dcls"));
    typeChecking(tree.child("statements"));
}
```
You need to make sure that every subtree that needs to be type-checked is actually visited.

Easy way: just visit every node (but this is inefficient).

Faster but harder way: explicitly recurse on the subtrees that need to be checked.

For example, for the main rule:

```java
if(tree.lhs == "main") {
    // check the second dcl and the return expr
    // now recurse on the parts that need checking: the dcls and statements
    typeChecking(tree.child("dcls"));
    typeChecking(tree.child("statements"));
}
```

For the dcls rules:

```java
if(tree.lhs == "dcls") {
    // check the type of the derived dcl
    // now recurse on the rest of the dcls
    typeChecking(tree.child("dcls"));
}
```
The End