Dealing with Syntax Errors

• One purpose of the parser is to filter out errors that show up in parsing.
• Later stages should not have to deal with possibility of malformed constructs.
• Parser must identify error so programmer knows what to correct.
• Parser should recover so that processing can continue (and other errors found).
• Parser might even correct error (e.g., PL/C compiler could "correct" some Fortran programs into equivalent PL/1 programs!).

Identifying Errors

• All of the valid parsers we’ve seen identify syntax errors “as soon as possible.”
• Valid prefix property: all the input that is shifted or scanned is the beginning of some valid program.
• … But the rest of the input might not be.
• So in principle, deleting the lookahead (and subsequent symbols) and inserting others will give a valid program.

Automating Recovery

• Unfortunately, best results require using semantic knowledge and hand tuning.
  - E.g., a[i].y = 5 might be turned to a[i]y = 5 if a is statically known to be a list, or a(i).y = 5 if a function.
• Some automatic methods can do an OK job that at least allows parser to catch more than one error.

Bison’s Technique

• The special terminal symbol error is never actually returned by the lexer.
• Gets inserted by parser in place of erroneous tokens.
• Parsing then proceeds normally.

Example of Bison’s Error Rules

• Suppose we want to throw away bad statements and carry on

```plaintext
stmt : whileStmt  
  | ifStmt  
  | ...  
  | error NEWLINE
;```

Response to Error

• Consider erroneous text like
  
  if x y: ...

  When parser gets to the y, will detect error.

  Then pops items off parsing stack until it finds a state that allows a shift or reduction on 'error' terminal

  Does reductions, then shifts 'error'.

  Finally, throws away input until it finds a symbol it can shift after 'error'

Error Response, contd.

• So with our example:

  stmt : whileStmt |
  ifStmt  | ...
  | error NEWLINE 

  We see 'y', throw away the 'if x', so as to be back to where a stmt can start.

  Shift 'error' and away more symbols to NEWLINE. Then carry on.

Of Course, It's Not Perfect

• "Throw away and punt" is sometimes called "panic-mode error recovery"

• Results are often annoying.

  For example, in our example, there's an INDENT after the NEWLINE, which doesn't fit the grammar and causes another error.

  Bison compensates in this case by not reporting errors that are too close together

  But in general, can get cascade of errors.

On to Static Semantics

• Lexical analysis
  - Produces tokens
  - Detects & eliminates illegal tokens

• Parsing
  - Produces trees
  - Detects & eliminates ill-formed parse trees

• Static semantic analysis
  - Produces "decorated tree" with additional information attached
  - Detects & eliminates remaining static errors

Static vs. Dynamic

• The term static used to indicate properties that the compiler can determine without considering any particular execution.

  - E.g., in
    
    def f(x) : x * 1

    Both uses of x refer to same variable

  - Dynamic properties are those that depend on particular executions in general. E.g., will x = x/y cause arithmetic exception.

Tasks of the Semantic Analyzer

• Find the declaration that defines each identifier instance

• Determine the static types of expressions

• Perform re-organizations of the AST that were inconvenient in parser, or required semantic information

• Detect errors and fix to allow further processing
Typical Semantic Errors: Java, C++

- **Multiple declarations**: A variable should be declared (in the same region) at most once.
- **Undeclared variable**: A variable should not be used before being declared.
- **Type mismatch**: Type of the left-hand side of an assignment should match the type of the right-hand side.
- **Wrong arguments**: Methods should be called with the right number and types of arguments.

A sample semantic analyzer

- Works in two phases
  - I.e., it traverses the AST created by the parser:
    1. For each declarative region in the program:
       - Process the declarations:
         - Add new entries to the symbol table and report any variables that are multiply declared.
         - Process the statements:
           - Find uses of undeclared variables, and update the "ID" nodes of the AST to point to the appropriate symbol-table entry.
    2. Process all of the statements in the program again:
       - Use the symbol-table information to determine the type of each expression, and to find type errors.

Symbol Table = set of entries

- **Purpose**:
  - Keep track of names declared in the program.
  - Names of:
    - Variables, classes, fields, methods.
- **Symbol table entry**:
  - Associates a name with a set of attributes, e.g.:
    - Kind of name (variable, class, field, method, etc).
    - Type (int, float, etc).
    - Nesting level.
    - Memory location (i.e., where will it be found at runtime).

Scoping

- Symbol table design influenced by what kind of scoping is used by the compiled language.
- **Scope of a declaration**: Section of text where it applies.
- **Declarative region**: Section of text that bounds scopes of declarations (we'll say "region" for short).
- In most languages, the same name can be declared multiple times:
  - If its declarations occur in different declarative regions, and/or
  - Involve different kinds of names.

Scoping: example

- Java: can use same name for:
  - A class.
  - Field of the class.
  - A method of the class, and
  - A local variable of the method.
- Legal Java program:

```java
class Test {
    int Test;
    Test() { double Test; }
}
```

Scoping: overloading

- Java and C++ (but not in Pascal, C, or Python):
  - Can use the same name for more than one method.
  - As long as the number and/or types of parameters are unique.

```java
int add(int a, int b);
float add(float a, float b);
```
Scoping: general rules

- The scope rules of a language:
  - determine which declaration of a named object corresponds to each use of the object.
  - i.e., scoping rules map uses of objects to their declarations.
- C++ and Java use static scoping:
  - mapping from uses to declarations is made at compile time.
  - C++ uses the "most closely nested" rule:
    - a use of variable x matches the declaration with the most closely enclosing scope.
    - a deeply nested variable x hides x declared in an outer region.
  - in Java:
    - inner regions cannot define variables defined in outer regions.

Scope levels

- In Java, each function has two or more declarative regions:
  - one for the parameters,
  - one for the function body,
  - and possibly additional regions in the function
    - for each for loop and
    - each nested block (delimited by curly braces).
- In Pyth, each function has one per function (possibly plus more for nested functions).

Example (assume C++ rules)

```cpp
void f(int k) { // k is a parameter
  int k = 0; // also a local variable (not legal in Java)
  while (k) {
    int k = 1; // another local var, in a loop (not ok in Java)
  }
  // the outermost region includes just the name "f", and
  // function f itself has three (nested) regions:
  // 1. The outer region for f just includes parameter k.
  // 2. The next region is for the body of f, and includes the variable k
  //    that is initialized to 0.
  // 3. The innermost region is for the body of the while loop, and
  //    includes the variable k that is initialized to 1.
}
```

Dynamic scoping

- Not all languages use static scoping.
  - Original Lisp, APL, and Snobol use dynamic scoping.
  - Dynamic scoping:
    - A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the most-recently-called still active function.
    - With this rule, difficult for compiler to determine much about identifiers.

Example

- For example, consider the following code:
  ```cpp
  void main() { f1(); f2(); }
  void f1() { int x = 10; g(); }
  void f2() { String x = "hello"; f3(); g(); }
  void f3() { double x = 30.5; }
  void g() { print(x); }
  ```
  - With static scoping, illegal.
  - With dynamic scoping, prints 10 and hello.

Used before declared?

- Can names be used before they are defined?
  - Java: a method or field name can be used before the definition appears; not true for a variable.
  - In Pyth, almost anything can be used before declaration, where syntactically possible.
Simplification

- From now on, assume that our language:
  - uses static scoping
  - requires that all names be declared before they are used
  - does not allow multiple declarations of a name in the same region
    - even for different kinds of names
  - does allow the same name to be declared in multiple nested regions
    - but only once per region
  - uses the same region for a method’s parameters and for the local variables declared at the beginning of the method

- Rules in Project 3 will differ!

Symbol Table Implementations

- In addition to the above simplification, assume that the symbol table will be used to answer two questions:
  1. Given a declaration of a name, is there already a declaration of the same name in the current region?
     - i.e., is it multiply declared?
  2. Given a use of a name, to which declaration does it correspond (using the “most closely nested” rule), or is it undeclared?

Symbol Table is Just Means to an End

- The symbol table is only needed to answer those two questions, i.e.
  - once all declarations have been processed to build the symbol table,
  - and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry,
  - then the symbol table itself is no longer needed
    - because no more lookups based on name will be performed

Decorating a Tree

- Program:
  \[
  \text{int } y = 17; \\
  \text{return } g(y); \\
  \]

- Idea: decorate tree with type, declaration data.

What operations do we need?

- Essentially, we need a data structure like environment diagrams in CS61A, minus dynamic information (i.e., variable values).
- So we will need to:
  1. Look up a name in the current declarative region only to check if it is multiply declared
  2. Look up a name in the current and enclosing regions
     - to check for a use of an undeclared name, and
     - to link a use with the corresponding symbol-table entry
  3. Insert a new name into the symbol table with its attributes.
  4. Do what must be done when entering a new region.
  5. Do what must be done when leaving a region.
Two possible symbol table implementations

1. a list of tables
2. a table of lists

• For each approach, we will consider
  - what must be done when entering and exiting a region,
  - when processing a declaration, and
  - when processing a use.
• Simplification:
  - assume each symbol-table entry includes only:
    • the symbol name
    • its type
    • the nesting level of its declaration

Example:

```c
void f(int a, int b) {
  double x;
  while (...) {
    int x, y; ...
  }
}
void g() { f(); }
```

• After processing declarations inside the while loop:

```
x: int, 3
y: int, 3
x: int, 2
y: int, 2
c: double, 2
```

Method 1: List of Dictionaries

• The idea:
  - symbol table = a list of dictionaries,
  - one dictionary for each currently visible region.
• When processing a declarative region S:

```
  front of list
  declarations made in S
  declarations made in regions that enclose S
  end of list
```

List of Dictionaries: The Operations

1. On entry to a declarative region:
   • increment the current level number and add a new empty dictionary to the front of the list.
2. To process a declaration of x:
   • look up x in the first dictionary in the list.
     • If it is there, then issue a "multiply declared variable" error;
     • otherwise, add x to the first table in the list.
3. To process a use of x:
   • look up x starting in the first dictionary in the list;
     • if it is not there, then look up x in each successive dictionary in the list.
     • if it is not in any dictionary then issue an "undeclared variable" error.
4. On leaving a region,
   • remove the first dictionary from the list and decrement the current level number.

Class Members

• For each class, associate a dictionary containing entries for each member.
• So given an expression such as x.clear(), we
  - find declaration for x in current dictionary
  - find type of x from its declaration, and
  - look up clear in dictionary associated with x's type.
The running times for each operation:

1. Region entry:
   - time to initialize a new, empty dictionary;
   - probably proportional to the size of the dictionary.
2. Process a declaration:
   - using hashing, constant expected time (O(1)).
3. Process a use:
   - using hashing to do the lookup in each dictionary in the list,
     the worst-case time is O(depth of nesting), when every table
     in the list must be examined.
4. Region exit:
   - time to remove a dictionary from the list, which should be
     O(1) if garbage collection is ignored.

Method 2: Dictionary of Lists

- the idea:
  - when processing a region, S, the structure of the
    symbol table is:

Definition

- there is just one big dictionary, containing an
  entry for each variable for which there is
  - some declaration in region S or
  - in a region that encloses S.
- Associated with each variable is a list of
  symbol-table entries.
  - The first list item corresponds to the most closely
    enclosing declaration;
  - the other list items correspond to declarations in
    enclosing regions.

Example

```c
void f(int x) {
  double x;
  while (...) { int x, y; ... }
  void g() { f(); }
}
```

- After processing the declarations inside the
  while loop:

Nesting level information is crucial

- The level-number attribute stored in each list
  item enables us to determine whether the
  most closely enclosing declaration was made
  - in the current region or
  - in an enclosing region.

Dictionary of lists: the operations

1. On region entry:
   - increment the current level number.
2. To process a declaration of x:
   - look up x in the symbol table.
     - If x is there, fetch the level number from the first list
       item.
     - If that level number = the current level then issue a
       “multiply declared variable” error;
     - otherwise, add a new item to the front of the list with
       the appropriate type and the current level number.
1. To process a use of \( x \):
   - look up \( x \) in the symbol table.
   - If it is not there, then issue an "undeclared variable" error.

2. On region exit:
   - scan all entries in the symbol table, looking at the first item on each list. If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry). Finally, decrement the current level number.

---

**Running times**

1. Scope entry:
   - time to increment the level number, \( O(1) \).

2. Process a declaration:
   - using hashing, constant expected time \( O(1) \).

3. Process a use:
   - using hashing, constant expected time \( O(1) \).

4. Scope exit:
   - time proportional to the number of names in the symbol table (assuming we can find the all names in linear time).

---

**Type Checking**

- the job of the type-checking phase is to:
  - Determine the type of each expression in the program (each node in the AST that corresponds to an expression)
  - Find type errors
- The type rules of a language define
  - how to determine expression types, and
  - what is considered to be an error.
- The type rules specify, for every operator (including assignment),
  - what types the operands can have, and
  - what is the type of the result.

---

**Type Errors**

- The type checker must also
  1. find type errors having to do with the context of expressions,
     - e.g., the context of some operators must be boolean,
     2. type errors having to do with method calls.
- Examples of the context errors:
  - the condition of an if not boolean (Java)
  - type of returned value not function’s return type
- Examples of method errors:
  - calling something that is not a method
  - calling a method with the wrong number of arguments
  - calling a method with arguments of the wrong types