Syntax-directed Translation

- Using syntax to drive translation of code
  - Type checking
  - Code generation
  - Symbol tables
  - Type checking

Motivation: parser as a translator

Syntax-directed translation

<table>
<thead>
<tr>
<th>stream of tokens</th>
<th>parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax + translation rules (typically hardcoded in the parser)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASTs, or assembly code</td>
</tr>
</tbody>
</table>
Mechanism of syntax-directed translation

- Syntax-directed translation is done by extending the CFG
  - A translation rule is defined for each production

  Given
  \[ X \rightarrow d \, A \, B \, c \]
  - The translation of \( X \) is defined in terms of:
    - Translation of nonterminals \( A, B \)
    - Values of attributes of terminals \( d, c \)
    - Constants

To translate an input string:

- Build the parse tree.
- Working bottom-up
  - Use the translation rules to compute the translation of each nonterminal in the tree

Result: The translation of the string is the translation of the parse tree's root nonterminal.

Why bottom up?

- A nonterminal's value may depend on the value of the symbols on the right-hand side,
- So translate a non-terminal node only after children translations are available.

Example 1: Arithmetic expression to its value

Syntax-directed translation:
- The CFG translation rules

<table>
<thead>
<tr>
<th>Production</th>
<th>Translation Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp \rightarrow exp + term</td>
<td>exp.trans = exp.trans + term.trans</td>
</tr>
<tr>
<td>exp \rightarrow term</td>
<td>exp.trans = term.trans</td>
</tr>
<tr>
<td>term \rightarrow term * factor</td>
<td>term.trans = term.trans * factor.trans</td>
</tr>
<tr>
<td>term \rightarrow factor</td>
<td>term.trans = factor.trans</td>
</tr>
<tr>
<td>factor \rightarrow INTLITERAL</td>
<td>factor.trans = INTLITERAL.value</td>
</tr>
<tr>
<td>factor \rightarrow ( exp  )</td>
<td>factor.trans = exp.trans</td>
</tr>
</tbody>
</table>
Example 1: Java Cup

Syntax-directed translation:

the CFG translation rules

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp}.\text{e}_1 + \text{term}.\text{t}_2 \quad \text{Result} = \text{e}_1 + \text{e}_2 \\
\text{exp} & \rightarrow \text{term}.\text{t}_1 \quad \text{Result} = \text{t}_1 \\
\text{term} & \rightarrow \text{term}.\text{t}_1 \ast \text{factor}.\text{f}_2 \quad \text{Result} = \text{t}_1 \ast \text{t}_2 \\
\text{term} & \rightarrow \text{factor}.\text{f}_1 \quad \text{Result} = \text{f}_1 \\
\text{factor} & \rightarrow \text{INTLITERAL}.i \quad \text{Result} = \text{value}.i \\
\text{factor} & \rightarrow (\text{exp}.\text{e}_1) \quad \text{Result} = \text{e}_1
\end{align*}
\]

Example 1 (cont)

Input: \(2 \ast (4 + 5)\)

Annotated Parse Tree

Example 2: Compute the type of an expression

\[
\begin{align*}
\text{exp} \rightarrow \text{exp} \ast \text{exp} & \quad \text{if } (\text{exp}_2.\text{trans} == \text{INT} \text{ and } (\text{exp}_3.\text{trans} == \text{INT}) \text{ then } \text{exp}_1.\text{trans} = \text{INT} \\
& \quad \text{else } \text{exp}_1.\text{trans} = \text{ERROR} \\
\text{exp} \rightarrow \text{exp} \ast \text{exp} & \quad \text{if } (\text{exp}_2.\text{trans} == \text{BOOL} \text{ and } (\text{exp}_3.\text{trans} == \text{BOOL}) \text{ then } \text{exp}_1.\text{trans} = \text{BOOL} \\
& \quad \text{else } \text{exp}_1.\text{trans} = \text{ERROR} \\
\text{exp} \rightarrow \text{exp} == \text{exp} & \quad \text{if } (\text{exp}_2.\text{trans} == \text{exp}_3.\text{trans}) \text{ and } (\text{exp}_2.\text{trans} != \text{ERROR}) \text{ then } \text{exp}_1.\text{trans} = \text{BOOL} \\
& \quad \text{else } \text{exp}_1.\text{trans} = \text{ERROR} \\
\text{exp} \rightarrow \text{true} & \quad \text{exp}.\text{trans} = \text{BOOL} \\
\text{exp} \rightarrow \text{false} & \quad \text{exp}.\text{trans} = \text{BOOL} \\
\text{exp} \rightarrow \text{int} & \quad \text{exp}.\text{trans} = \text{INT} \\
\text{exp} \rightarrow (\text{exp}) & \quad \text{exp}.\text{trans} = \text{exp}_1.\text{trans}
\end{align*}
\]
Building Abstract Syntax Trees

- Examples so far, streams of tokens translated into
  - integer values, or
  - boolean values

- Translating into ASTs is not very different

AST vs Parse Tree

- AST is a condensed form of a parse tree
  - operators appear at internal nodes, not at leaves.
  - "Chains" of single productions are collapsed.
  - Lists are "flattened".
  - Syntactic details are omitted
    - e.g., parentheses, commas, semi-colons

- AST is a better structure for later compiler stages
  - omits details having to do with the source language,
  - just contains information about the essential structure of the program.

Example: \( 2 \times (4 + 5) \) parse tree vs AST
Definitions of AST nodes

```java
class ExpNode { }

class IntLitNode extends ExpNode {
    public IntLitNode(int val) { ... }
}

class PlusNode extends ExpNode {
    public PlusNode(ExpNode e1, ExpNode e2) {
        ...
    }
}

class TimesNode extends ExpNode {
    public TimesNode(ExpNode e1, ExpNode e2) {
        ...
    }
}
```

AST-building translation rules

```plaintext
exp -> exp + term  \rightarrow  \text{newPlusNode}(exp.trans, term.trans)
exp -> term          \rightarrow  term.trans
term -> term * factor \rightarrow  \text{newTimesNode}(term.trans, factor.trans)
term -> factor       \rightarrow  factor.trans
factor -> INTLITERAL \rightarrow  \text{newIntLitNode}(\text{INTLITERAL}.value)
factor -> ( exp )    \rightarrow  factor.trans = exp.trans
```

Syntax-Directed Translation

- Syntax-directed translation is computed bottom-up.
- With parsing stack, have **semantic stack**:
  - Holds nonterminals' translations
  - When the parse is finished, it will hold just one value: the translation of the root nonterminal (which is the translation of the whole input).
How does semantic stack work?

- Intuitively, semantic stack parallels the parser stack.
- Parser stack has grammar symbols (states); semantic stack has values (e.g., tree nodes).
- In reality, values are pushed/popped onto/off the semantic stack by adding actions to the grammar rules.

Rules

- action for $X \rightarrow Y_1 Y_2 ... Y_n$ is pushed onto the (normal) stack when the derivation step $X \rightarrow Y_1 Y_2 ... Y_n$ is made, but
- the action is performed only after complete derivations for all of the $Y$'s have been carried out.

Semantic rules

- Abstract syntax trees can be constructed in bottom-up fashion.
- Consider evaluation using bottom-up parser.

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E :: T :: i$</td>
<td>Result = new PlusNode(e1, e2)</td>
</tr>
<tr>
<td>$E \rightarrow E :: T :: i$</td>
<td>Result = new MinusNode(e1, e2)</td>
</tr>
<tr>
<td>$E \rightarrow T :: t$</td>
<td>Result = t</td>
</tr>
<tr>
<td>$T \rightarrow (E :: e)$</td>
<td>Result = e</td>
</tr>
<tr>
<td>$T \rightarrow id :: i$</td>
<td>Result = i</td>
</tr>
<tr>
<td>$T \rightarrow IntLiteral :: i$</td>
<td>Result = new IntLitNode(i.lineno, i.value)</td>
</tr>
<tr>
<td>$id \rightarrow ID :: i$</td>
<td>Result = new IdNode(i.lineno, i.idval)</td>
</tr>
</tbody>
</table>
Semantic Analysis

- How to build symbol tables
- How to use them to find
  - multiply-declared and
  - undeclared variables.
- How to perform type checking

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
    - e.g.: main$ ();
- Parsing
  - Detects inputs with ill-formed parse trees
    - e.g.: missing semicolons
- Semantic analysis
  - Last “front end” phase
  - Catches all remaining errors
Introduction

- typical semantic errors:
  - multiple declarations: a variable should be declared (in the same scope) 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.

An sample semantic analyzer

- works in two phases
  - i.e., it traverses the AST created by the parser:
  - For each scope 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.
  - Process all of the statements in the program again,
    - use the symbol-table information to determine the type of each expression and find type errors.

Symbol Table = Map keys -> values

- purpose:
  - keep track of names declared in the program
  - names of
    - variables, classes, fields, methods, etc.
- 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
- In most languages, the same name can be declared multiple times
  - if its declarations occur in different scopes, and/or
  - involve different kinds of names.

Scoping: example

- Java: can use same name for
  - a class,
  - instance variables,
  - a method of the class, and
  - a local variable of the method
- legal Java program:

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

Scoping: overloading

- Java and C++ (but not in Pascal or C):
  - 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 in the most closely enclosing scope
    - such that the declaration precedes the use.

Scope levels

- Each function has two or more scopes:
  - one for the parameters,
  - one for the function body,
  - and possibly additional scopes in the function
    - for each for loop and
    - each nested block (delimited by curly braces)

Example

```c
void f( int k ) {  // k is a parameter
    int k = 0;     // also a local variable
    while (k) {
        int k = 1;  // another local variable, in a loop
    }
}
```

- the outmost scope includes just the name "f", and
- function f itself has three (nested) scopes:
  - The outer scope for f just includes parameter k.
  - The next scope is for the body of f, and includes the variable k that is initialized to 0.
  - The innermost scope 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.
- 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.

Attributes about symbols

- Variables - textual name, type, nesting level, offset in storage
- Methods or procedures - number and type of arguments, type of return value, nesting level
- Structures - number of dimensions, type of index, type of element
- Call by reference parameters and pointers - possible aliases

Symbol table structure affected by scoping

- We will focus on static scoping - dynamic scoping seldom used except in rare conditions e.g., exception handling in Java.
- Can a name be used before they are defined?
- Different rules for static scoping
  - Everything must be declared before used
  - Java: a method or instance variable name can be used before the definition appears,
    - not true for a local variable (defined within method)
  - Most closely nested rule
  - What is a scope? Loop, block, parameters
Example

class Test {
    void f() {
        val = 0;
        // instance variable val has not yet been declared -- OK
        g();
        // method g has not yet been declared -- OK
        x = 1;
        // var x has not yet been declared -- ERROR!
        int x;
    }
    void g() {
    }
    int val;
}

Simplification

- From now on, assume that our language (rules for Simple)
  - uses static scoping
  - requires that all names be declared before used
  - does not allow multiple declarations of a name in the same scope (even for different kinds of names)
  - does allow the same name to be declared in multiple nested scopes
  - uses the same scope for a method’s parameters and for the local variables declared at the beginning of the method
- This will make your life in P4 much easier!

Symbol Table Implementations

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

- The symbol table is only needed to answer those two questions, i.e.
  - after 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 functions are no longer needed
    - because no more lookups based on name will be performed

What operation do we need?

- Given the above assumptions, we will need:
  - Look up a name in the current scope only
    - to check if it is multiply declared
  - Look up a name in the current and enclosing scopes
    - to check for a use of an undeclared name, and
    - to link a use with the corresponding symbol-table entry
  - Insert a new name into the symbol table with its attributes.
  - Change symbol table when a new scope is entered.
  - Change symbol table when a scope is exited.

Two possible symbol table implementations

- a list of tables
- a table of lists
- For each approach, we will consider what must be done
  - when entering and exiting a scope,
  - 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
Method 1: List of Hash tables

- The idea:
  - symbol table = a list of hash tables,
  - one hash table for each currently visible scope.

- When processing a scope $S$:
  - front of list
  - end of list

  declarations made in $S$

  declarations made in
  scopes that enclose $S$

Example:

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

After processing declarations inside the while loop:

format: var: type, nesting level

- $x$: int, 3
- $y$: int, 3
- $z$: double, 2
- $f$: (int, int) → void, 1

List of hash tables: the operations

- On scope entry:
  - increment the current level number
  - add a new empty hash table to the front of the list.

- To process a declaration of $x$:
  - look up $x$ in the first table in the list - hash on
    name:
    - $h(x)$
    - if it is there, then issue a "multiply declared variable"
      error;
    - otherwise, add $x$ to the first table in the list.
... continued

- To process a use of x:
  - look up x starting in the first table in the list;
  - if it is not there, then look up x in each successive table
    in the list.
  - if it is not in any table then issue an "undeclared variable" error.

- On scope exit,
  - remove the first table from the list and
decrement the current level number.

Remember for our language Simple

- method names belong into the hash table for
  the outermost scope
  - not into the same table as the method's variables

- For example, in the example above:
  - method name f is in the symbol table for the
    outermost scope
  - name x is not in the same scope as parameters a and b, and variable x.
  - This is so that when the use of name f in method
g is processed, the name is found in an enclosing
  scope's table.

The running times for each
operation:

- Scope entry:
  - time to initialize a new, empty hash table;
    probably proportional to the size of the hash table.
- Process a declaration:
  - using hashing, constant expected time (O(1)).
- Process a use:
  - using hashing to do the lookup in each table in
    the list, the worst-case time is O(depth of nesting)
- Scope exit:
  - time to remove a table from the list, if storage
    reclamation is ignored O(1)
Consider another language

C++ does not use exactly the scoping rules that we have been assuming.
- In particular, C++ does allow a function to have both a parameter and a local variable with the same name
- any uses of the name refer to the local variable

Questions 1: Consider the following code. What would the symbol table look like after processing the declarations in the body of \( f \) under:
- the scoping rules we have been assuming
- C++ scoping rules

```cpp
void g(int x, int a) { }
void f(int x, int y, int z) { int a, b, x; ... }
```

... continued

Question 2:
- Which of the four operations described above
  - scope entry,
  - process a declaration,
  - process a use,
  - scope exit
would change (and how) if the following rules for name reuse were used instead of C++ rules:
- the same name can be used within one scope as long as the uses are for different kinds of names, and
- the same name cannot be used for more than one variable declaration in a nested scope

Method 2: Hash table of Lists

- the idea:
  - when processing a scope \( S \), the structure of the symbol table is:
Definition

- there is just one big hash table, containing an entry for each variable for which there is
  - some declaration in scope S or
  - in a scope 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 scopes.

Example

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

```
f: int ➔ void, 1
  x: int, 2
  y: int, double, 2
  z: int, 3
```

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 scope or
  - in an enclosing scope.
Hash table of lists: the operations

- On scope entry:
  - increment the current level number.
- 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.

... process use of x:

- look up x in the symbol table.
  - if it is not there, then issue an "undeclared variable" error.
- On scope exit:
  - must restore table to previous nesting level
  - sometimes must keep entry around
- Three ways:
  - 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.

Exiting from scope - continued

- use extra pointer in each element to link all items of the same scope
  3. use a stack that has entries for each scope

Pop entry and delete from hash table - keep stack entries around
What if collisions?

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

- After processing the declarations inside the while loop: assume x and f collide.

<table>
<thead>
<tr>
<th>x:</th>
<th>int, 2</th>
<th>f:</th>
<th>void, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:</td>
<td>int, 3</td>
<td>y:</td>
<td>double, 2</td>
</tr>
<tr>
<td>z:</td>
<td>int, 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running times

- Scope entry:
  - time to increment the level number, $O(1)$.
- Process a declaration:
  - using hashing, constant expected time ($O(1)$).
- Process a use:
  - using hashing, constant expected time ($O(1)$).
- Scope exit:
  - time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable).

Another example

- Assume that the symbol table is implemented using a hash table of lists.
- How does the symbol table changes as each declaration in the following code is processed.

```c
void g(int x, int a) {
    double d;
    while (...) {
        int d, w;
        double x, b;
        if (...) { int a, b, c; }
    }
    while (...) { int x, y, z; }
```
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.

Types

- Basic types (primitives): integers, characters, booleans
- Input/output streams - pointer to buffer
- Aggregate - arrays
- User defined - classes, enumerated

Example

- Both C++ and Java allow the addition of an int and a double, and the result is of type double.
- However,
  - C++ also allows a value of type double to be assigned to a variable of type int,
  - Java considers that an error.