Any compiler must perform two major tasks:

- **Analysis** of the source program
- **Synthesis** of a machine-language program

**The Structure of a Compiler**

1. **Compiler**
   - **Analysis**
   - **Synthesis**

**Semantic Processing**

- Semantic routines interpret meaning based on syntactic structure of input (modern compilers do this)
- This makes the compilation syntax-directed
- Semantic routines finish the analysis
  - Verify static semantics are followed
  - Variables declared, compatible operands (type and #), etc.
- Semantic routines also start the synthesis
  - Generate either IR or target machine code
- The semantic action is attached to the productions (or sub trees of a syntax tree).

**Abstract Syntax Tree**

- 1st step in semantic processing is to build a syntax tree representing input program
- Don't need literal parse tree
  - Intermediate nodes for precedence and associativity
  - e-rules
  - Just enough info to drive semantic processing
  - Or even recreate input
- Semantic processing performed by traversing the tree 1 or more times
  - Attributes attached to nodes aid semantic processing

**Compiler Stages**

- Source Code
  - Scanner
  - Tokens
  - Parser
  - Syntax Tree
  - Semantic Analyzer
  - Optimizer
  - Intermediate Code
  - Code Generator
  - Target Code

- Analysis
  - Symbol and Attribute Tables
  - (Used by all Phases of The Compiler)
  - Target machine code

- Synthesis
  - Code Generator
  - Target machine code
7.1.1 Using a Syntax Tree Representation of a Parse (1)

- **Parsing:**
  - build the parse tree
  - Non-terminals for operator precedence and associativity are included.

- **Semantic processing:**
  - build and decorate the Abstract Syntax Tree (AST)
  - Non-terminals used for ease of parsing may be omitted in the abstract syntax tree.

- Abstract syntax tree for $Y:=3*X+I$

- Abstract syntax tree for $Y:=3*X+I$ with initial values

---

7.1.1 Using a Syntax Tree Representation of a Parse (2)

- Semantic routines traverse (post-order) the AST, computing attributes of the nodes of AST.

- Initially, only leaves (i.e., terminals, e.g., const, id) have attributes.

- Abstract syntax tree for $Y:=3*X+I$ with propagated values
7.1.1 Using a Syntax Tree Representation of a Parse (3)

- The attributes are then propagated to other nodes using some functions, e.g.
  - build symbol table
  - attach attributes of nodes
  - check types, etc.

- **bottom-up / top-down** propagation

7.1.1 Using a Syntax Tree Representation of a Parse (4)

- After attribute propagation is done, the tree is decorated and ready for code generation, use another pass over the decorated AST to generate code.

- Actually, these can be combined in a single pass
  - Build the AST
  - Decorate the AST
  - Generate the target code

- What we have described is essentially the **Attribute Grammars (AG)** (Details in chap. 14)

---

**Static Semantic Checks**

- Static semantics can be checked at compile time
- Check only propagated attributes
- Type compatibility across assignment
  - `i = 5.2` illegal
  - `i = 5` legal
- Use attributes and structure
  - Correct number and types of parameters
  - Procedure call `a float k, int i, float b` at `i`:
    - `float i` legal
    - `call exp(2.5)` illegal
    - `call test(3.3)` illegal

**Dynamic Semantic Checks**

- Some checks can't be done at compile time
  - Array bounds, arithmetic errors, valid addresses of pointers, variables initialized before use.
- Some languages allow explicit dynamic semantic checks
  - i.e. `assert denominator not = 0`

- These are handled by the semantic routines inserting code to check for these semantics

- Violating dynamic semantics result in exceptions

---

**Translation**

- Translation task uses attributes as data, but it is driven by the structure
- Translation output can be several forms
  - Machine code
  - Intermediate representation
  - Decorated tree itself
  - Sent to optimizer or code generator

---

**Compiler Organization**

- one-pass compiler
  - Single pass used for both analysis and synthesis
  - Scanning, parsing, checking, & translation all interleaved
  - No explicit IR generated
  - Semantic routines must generate machine code
  - Only simple optimizations can be performed
  - Tends to be less portable
7.1.2 Compiler Organization Alternatives (2)

- We prefer the code generator completely hides machine details and semantic routines are independent of machines.

- Can be violated to produce better code.
  - Suppose there are several classes of registers, each for a different purpose.
  - Better for register allocation to be performed by semantic routines than code generator since semantic routines have a broader view of the AST.

Compiler Organization

- Multi-pass analysis
  - Scan, then parse, then check declarations, then static semantics
  - Usually used to save space (memory usage or compiler)

- Multi-pass synthesis
  - Separate out machine dependence
  - Better optimization
  - Generate IR
  - Do machine independent optimization
  - Generate machine code
  - Machine dependent optimization

- Many complicated optimization and code generation algorithms require multiple passes
  - i.e. optimizations that need a more global view

7.1.3 Compiler Organization Alternatives (7)

- Multi-language and multi-target compilers
  - Components may be shared and parameterized.
  - Ex: Ada uses Diana (language-dependent IR)
  - Ex: GCC uses two IRs.
  - one is high-level tree-oriented
  - the other (RTL) is more machine-oriented

7.1.3 Single Pass (1)

- In Micro of chap 2, scanning, parsing and semantic processing are interleaved in a single pass.

  - (+) simple front-end
  - (+) less storage if no explicit trees
  - (-) immediately available information is limited since no complete tree is built.

Relationships
### 7.1.3 Single Pass (2)

- Each terminal and non-terminal has a semantic record.
- Semantic records may be considered as the attributes of the terminals and non-terminals.
  - **Terminals**
    - the semantic records are created by the scanner.
  - **Non-terminals**
    - the semantic records are created by a semantic routine when a production is recognized.
- Semantic records are transmitted among semantic routines via a semantic stack.

### 7.1.3 Single Pass (3)

- 1 pass = 1 post-order traversal of the parse tree
- Parsing actions ↔ build parse trees
- Semantic actions ↔ post-order traversal

### Chapter 6 - Semantic Analysis

- **Parser** verifies that a program is syntactically correct and constructs a syntax tree (or other intermediate representation).
- **Semantic analyzer** checks that the program satisfies all other static language requirements (is "meaningful") and collects and computes information needed for code generation.

---

**Important Semantic Information**

- **Symbol table**: collects declaration and scope information to satisfy "declaration before use" rule, and to establish data type and other properties of names in a program.
- **Data types and type checking**: compute data types for all typed language entities and check that language rules on types are satisfied.

**How to build the symbol table and check types:**

- Analyze the scope rules for the language and determine an appropriate table structure for maintaining this information.
- Analyze the type requirements and translate them into rules that can be applied recursively on a syntax tree.

**Theoretical framework for semantic analysis**

- **Focus on attributes**: computable properties of language constructs that are needed to satisfy language requirements and/or generate code.
- **Describe the computation of attributes using equations or algorithms**.
- **Associate these equations to grammar rules and/or kinds of nodes in a syntax tree**.

---
Analyze the structure of the equations to determine an order in which the attributes can be computed. (Tree traversals of syntax tree - preorder, postorder, inorder, or some combination of them.)

Such a set of equations as described is called an **attribute grammar**.

While much can be done without a formal framework, the formality of equations can help the process considerably.

Nevertheless, there is currently no tool in standard use that allows this process to be automated (languages differ too much in their requirements).

Example of an attribute grammar

**Grammar:**

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp} + \text{term} \\
\text{exp} & \rightarrow \text{exp} - \text{term} \\
\text{exp} & \rightarrow \text{term} \\
\text{term} & \rightarrow \text{term} \cdot \text{factor} \\
\text{term} & \rightarrow \text{factor} \\
\text{factor} & \rightarrow ( \text{exp} ) \\
\text{factor} & \rightarrow \text{number}
\end{align*}
\]

**Attribute Grammar:**

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{exp} \rightarrow \text{exp} + \text{term}</td>
<td>\text{exp}.\text{val} = \text{exp}.\text{val} + \text{term}.\text{val}</td>
</tr>
<tr>
<td>\text{exp} \rightarrow \text{exp} - \text{term}</td>
<td>\text{exp}.\text{val} = \text{exp}.\text{val} - \text{term}.\text{val}</td>
</tr>
<tr>
<td>\text{exp} \rightarrow \text{term}</td>
<td>\text{exp}.\text{val} = \text{term}.\text{val}</td>
</tr>
<tr>
<td>\text{term} \rightarrow \text{term} \cdot \text{factor}</td>
<td>\text{term}.\text{val} = \text{term}.\text{val} \cdot \text{factor}.\text{val}</td>
</tr>
<tr>
<td>\text{term} \rightarrow \text{factor}</td>
<td>\text{term}.\text{val} = \text{factor}.\text{val}</td>
</tr>
<tr>
<td>\text{factor} \rightarrow ( \text{exp} )</td>
<td>\text{factor}.\text{val} = \text{exp}.\text{val}</td>
</tr>
<tr>
<td>\text{factor} \rightarrow \text{number}</td>
<td>\text{factor}.\text{val} = \text{number}.\text{val}</td>
</tr>
</tbody>
</table>

Notes:

- Different instances of same nonterminal must be subscripted to distinguish them.
- Some attributes must have been precomputed (by scanner or parser), e.g. \text{number}.\text{val}.
- These particular attribute equations look a lot like a yacc specification, because they represent a **bottom-up** attribute computation.

A Second Example

**Grammar:**

\[
\begin{align*}
\text{decl} & \rightarrow \text{type} \text{var-list} \\
\text{type} & \rightarrow \text{int} | \text{float} \\
\text{var-list} & \rightarrow \text{id} , \text{var-list} | \text{id}
\end{align*}
\]

**Attribute Grammar:**

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{decl} \rightarrow \text{type} \text{var-list}</td>
<td>\text{var-list}.\text{dtype} = \text{type}.\text{dtype}</td>
</tr>
<tr>
<td>\text{type} \rightarrow \text{int}</td>
<td>\text{type}.\text{dtype} = \text{integer}</td>
</tr>
<tr>
<td>\text{type} \rightarrow \text{float}</td>
<td>\text{type}.\text{dtype} = \text{real}</td>
</tr>
<tr>
<td>\text{var-list} \rightarrow \text{id} , \text{var-list}</td>
<td>\text{id}.\text{dtype} = \text{var-list}.\text{dtype}</td>
</tr>
<tr>
<td>\text{var-list} \rightarrow \text{id}</td>
<td>\text{var-list}.\text{dtype} = \text{id}.\text{dtype}</td>
</tr>
</tbody>
</table>

Notes:

- Data type typically propagates **down** a syntax tree via declarations.
- No longer something yacc can handle directly.
- Such an attribute is called **inherited**, while bottom-up calculation is called **synthesized**.
- Syntax tree is a standard synthesized attribute computable by yacc; other attributes computed on the tree.
Dependency graph

- Indicates order in which attributes must be computed.
- Synthesized attributes always flow from children to parents, and can always be computed by a postorder traversal.
- Inherited attributes can flow any other way.
- L-attributed: a left-to-right traversal suffices to compute attributes. However, this may involve a combination of pre-order, inorder, and postorder traversal.

Data type dependencies (by grammar rule):

- \( \text{decl} \rightarrow \text{type} \text{ var-list} \):
  \[ \text{var-list}.\text{dtype} = \text{type}.\text{dtype} \]

- \( \text{var-list} \rightarrow \text{id}, \text{var-list} \):
  \[ \text{id}.\text{dtype} = \text{var-list}.\text{dtype} \]

- \( \text{var-list} \rightarrow \text{dtype} \text{ var-list} \):
  \[ \text{dtype}.\text{id} = \text{var-list}.\text{dtype} \]

L-attributed dependencies have three basic mechanisms:

(a) Inheritance from parent to siblings
(b) Inheritance from sibling to sibling via the parent
(c) Sibling inheritance via sibling pointers

Sample tree structure:

```c
typedef enum {decl, type, id} nodekind;
typedef enum {integer, real} typekind;
typedef struct treeNode
{ nodekind kind;
  struct treeNode * lchild, * rchild, * sibling;
  typekind dtype;
  /* for type and id nodes */
  char * name;
  /* for id nodes only */
} * SyntaxTree;
```

Sample tree instance:

String: `float x, y`

Tree:
```
  decl
    type
      id
        ( dtype = real )
       id
         (x) | y
```

Traversal code:

```c
void evalType(SyntaxTree t)
{
    switch (t->kind)
    {
    case decl:
        t->rchild->dtype = t->lchild->dtype;
        evalType(t->rchild);
        break;
    case id:
        if (t->sibling != NULL)
        {
            t->sibling->dtype = t->dtype;
            evalType(t->sibling);
        }
        break;
    } /* end switch */
} /* end evalType */
```
Attributes need not be kept in the syntax tree:

<table>
<thead>
<tr>
<th>GRAMMAR RULE</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| decl → type var-list | dtype = integer  
dtype = real  
insert(id, dtype)  
insert(id, dtype) |
| type → float | dtype = integer  
dtype = real  
insert(id, dtype)  
insert(id, dtype) |
| var-list → id , var-list | dtype = integer  
dtype = real  
insert(id, dtype)  
insert(id, dtype) |
| var-list → id | dtype = integer  
dtype = real  
insert(id, dtype)  
insert(id, dtype) |

Use a symbol table to store the type of each identifier

dtype is global

New traversal code:

```c
void evalDecl (SyntaxTree t)
{
    switch (t->kind)
    {
        case decl:
            dtype = t->lchild->dtype;
            evalType(t->rchild);
            break;
        case id:
            insert(t->name, dtype);
            if (t->sibling != NULL)
                evalType(t->sibling, dtype);
            break;
    } /* end switch */
} /* end evalDecl */
```

Even better, use a parameter instead of a global variable:

```c
void evalType (SyntaxTree t, typekind dtype)
{
    insert(t->name, dtype);
    if (t->sibling != NULL)
        evalType(t->sibling, dtype);
}
```

Note: inherited attributes can often be turned into parameters to recursive traversal functions, while synthesized attributes can be turned into returned values.

Alternative to a difficult inherited situation (not recommended):

**Theorem** (Knuth [1968]). Given an attribute grammar, all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.

Example:

New grammar for types:

```c
decl → var-list id
var-list → var-list id , type
type → int | float
```

New Tree for `float x, y` might be:

```
  type
   /
  float
   /
    x
    /
dtype ( = real )
  /
  id ( = real )
```

Our approach:

- Compute inherited stuff first (symbol table) in a separate pass
- Then type inference and type checking turns into a purely synthesized attribute computation, since all uses of names have their types already computed.
- Next:
  - Symbol table structure
  - Synthesized type rules
7.2.2 LR(1) - (1)
- **Semantic routines**
  - are invoked only when a structure is recognized.

- **LR parsing**
  - a structure is recognized when the RHS is reduced to LHS.

- Therefore, **action symbols** must be placed at the end.

**Ex:**

```
<stmt> -> if <exp> then <stmt> end
        | if <exp>, then <stmt> else <stmt> end
```

7.2.2 LR(1) - (2)
- **After shifting "if <exp>"**
  - The parser cannot decide which of #ifThen and #ifThenElse should be invoked.

- **cf. In LL parsing,**
  - The structure is recognized when a non-terminal is expanded.

7.2.2 LR(1) - (3)
- **However, sometimes we do need to perform semantic actions in the middle of a production.**

**Ex:**

```
<stmt> -> if <exp> then <stmt> end
        | if <exp>, then <stmt> else <stmt> end
```

**Solution:** Use two productions:

```
<stmt> -> #finishIf
        | #startIf <exp>
        | #stahIf <stmt>
```

7.2.2 LR(1) - (4)
- **Another problem**
  - What if the action is not at the end?

**Ex:**

```
<prog> -> #start begin <stmt> end
```

**Solution:** Introduce a new non-terminal.

```
<prog> -> <head> begin <stmt> end
        | <head> #start
```

7.2.3 Semantic Record Representation - (1)
- **Since we need to use a stack to store semantic records,**
  - all semantic records must have the same type.

  - variant record in Pascal
  - union type in C

**Ex:**

```
enum kind {OP, EXP, STMT, ERROR};
typedef struct {
  enum kind tag;
  union {
    op_rec_type   OP_REC;
    expr_rec_type EXP_REC;
    stmt_rec_type STMT_REC;
    ...
  }
} sem_rec_type;
```

7.2.3 Semantic Record Representation - (2)
- **How to handle errors?**

**Ex.**

- A semantic routine needs to create a record for each identifier in an expression.
  - What if the identifier is not declared?
  - The solution at next page…….
7.2.3 Semantic Record Representation - (3)

- Solution 1: make a bogus record
  This method may create a chain of meaningless error messages due to this bogus record.

- Solution 2: create an ERROR semantic record
  No error message will be printed when ERROR record is encountered.

- WHO controls the semantic stack?
  - action routines
  - parser

7.2.4 Action-controlled semantic stack - (1)

- Action routines take parameters from
  the semantic stack directly and push results onto the stack.

- Implementing stacks:
  1. array
  2. linked list

- Usually, the stack is transparent - any records in the stack may be accessed by the semantic routines.
  - (+) difficult to change

7.2.4 Action-controlled semantic stack - (2)

- Two other disadvantages:
  - (+) Action routines need to manage the stack.
  - (-) Control of the stack is distributed among action routines.
    - Each action routine pops some records and pushes 0 or 1 record.
    - If any action routine makes a mistake, the whole stack is corrupt.

  The solution at next page...........

7.2.5 Parser-controlled stack - (1)

- LR
  Semantic stack and parse stack operate in parallel [shifts and reduces in the same way].

- Ex:
  - `<stmt>` -> `if <exp> then <stmt> end`

- Ex:
  - YACC generates such parser-controlled semantic stack.
    - `<exp>` -> `<exp> + <term>`
    - `{ $$.value=$1.value+$3.value; }

7.2.5 Parser-controlled stack - (2)

- LL parser-controlled semantic stack
  Every time a production A -> B C D is predicted,

  Parse stack:

  Semantic stack:

  Need four pointers for the semantic stack (left, right, current, top).
However, when a new production $B \rightarrow EFG$ is predicted, the four pointers will be overwritten.

Therefore, create a new EOP record for the four pointers on the parse stack.

When EOP record appears on stack top, restore the four pointers, which essentially pops off records from the semantic stack.

An example at next page......

Note

All push() and pop() are done by the parser

Not by the action routines.

Semantic records

Are passed to the action routines by parameters.

Example

<primary> -> <exp>  #copy ($2,$$)

Semantic stack may grow very big.

Example

<stmt list> -> <stmt>  #reuse <stmt tail>

<stmt tail> -> <stmt>  #reuse <stmt tail>

<stmt tail> Evaluation

Parser-controlled semantic stack is easy with LR, but not so with LL.

Intermediate representation and code generation

Two possibilities:

1. (+) no extra pass for code generation
   (+) allows simple 1-pass compilation

2. (+) allows higher-level operations e.g. open block, call procedures.
IR vs Machine Code

- Generating machine code advantages:
  - No overhead of extra pass to translate IR
  - Conceptually simple compilation model

- Bottom line
  - IR valuable if optimization or portability is an important issue
  - Machine code much simpler

Forms of IR – Postfix Notation

- Concise
- Simple translation
- Useful for interpreters and target machines with a stack architecture
- Not particularly good for optimization or code generation

Example:

<table>
<thead>
<tr>
<th>Code</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b</td>
<td>ab+</td>
</tr>
<tr>
<td>a+b+c</td>
<td>abc+</td>
</tr>
<tr>
<td>(a+b)*c</td>
<td>ab+c*</td>
</tr>
<tr>
<td>a:=b<em>c+b</em>d</td>
<td>abc<em>b</em>d:=</td>
</tr>
</tbody>
</table>

Forms of IR – Three-Address Codes

- Virtual machine having operations with 3 operands, 2 sources, 1 destination
- Explicitly reference intermediates
  - More concise
  - Position dependency makes moving/removing triples hard
  - Such as during optimization

- Triples: op, arg1, arg2
- Quadruples: op, arg1, arg2, arg3

```
Float a,d; Int b,c;
a := b*c + b*d

(1) (MULTI, Addr(b), Addr(c), t1)
(2) (FLOAT, Addr(b), t2)
(3) (MULTF, t2, Addr(d), t3)
(4) (FLOAT, t1, t4)
(5) (ADDF, t4, t3, t5)
(6) (:=, t5, Addr(a))
```

- Can also add more detail, such as type or address.
- These forms translate input, other 3 forms transform it

Forms of IR – Tuples

- Tuples allow variable number of operands
- A generalization of quadruples

```
a := b*c + b*d

(1) (MULTI, Addr(b), Addr(c), t1)
(2) (FLOAT, Addr(b), t2)
(3) (MULTF, t2, Addr(d), t3)
(4) (FLOAT, t1, t4)
(5) (ADDF, t4, t3, t5)
(6) (:=, t5, Addr(a))
```

Forms of IR – Trees

- Syntax trees can also be used
- Directed cyclic graph (DAG) is an option
- Can use an abstract syntax tree
- More complex and more powerful

```
Ex: a := b*c + b*d
```

```plaintext
Ex. Ada uses Diana.
```

```
Tree Transformations for optimizations
```

```
Ex: Ada uses Diana.
```
Symbol Table
- Major data structure after syntax tree.
- An inherited attribute that may be kept globally.
- May be needed before semantic analysis (or some form of it, as in C), but makes sense to put off computing it until necessary.
- Stores declaration information using name as primary key.

Specific information stored in symbol table depends heavily on language, but generally includes:
- Data type
- Scope (see below)
- Size (bytes, array length)
- Potential or actual location information (addresses, offsets - see later)

One way to finesse the issue of what information to put into the table is to just keep pointers in the table that point to declaration nodes in the syntax tree. Then symbol table code doesn't need to be changed when changing the information, since it is stored in the node, not directly in the table. This is the approach taken in the TINY compiler, and should be carried over to C-Minus.

Scope Information
- Requires that symbol table have some kind of “delete” operation in addition to lookup and insert, since exiting a scope requires that declarations be removed from view (that is, lookups no longer find them, though they may still be referenced elsewhere).
- Delete operation should not in general re-process individual declarations: exitScope() should do them all in O(1).

C has simple scope structure:
- All names must be declared before use (although multiple declarations are possible).
- Scopes are nested in a stack-like fashion, and cannot be re-entered after exit (simple delete is possible).
- Scope information can be kept simply as a number: the nesting level (needed during semantic analysis because redeclaration in same scope is illegal in C).

Example:
```c
typedef int z;
int y;
/* this is legal C! */
void x(double x)
{
    char* x;
    { char x;
    }
}
```

"external" (global) scope: nestLevel 0
nestlevel 1 begins with params
nestlevel 2 begins with function body
nestlevel 3
Not all compilers get it right that parameters have a separate scope from the function body in C. But gcc does:

```c
C:\classes\cs153\f02>gcc -c scope.c
scope.c: In function `x':
scope.c:6: warning: declaration of `x' shadows a parameter
```

At least all names occupy a single “namespace” in C, so one symbol table is enough (compare to Java).

Java has 5 “namespaces”, depending on type of declaration:

```java
package A; // legal Java!!!
class A
{ A A(A A)
  { A:
    for(;;)
    { if (A.A(A) == A) break A; }
    return A;
  }
}
```

Further complication in Java: local redeclaration even in nested scopes is illegal:

```java
class A
{ A A(A A)
  { for(;;)
    { A A; // oops, now illegal!
      if (A.A(A) == A) break;
    }
    return A;
  }
}
```

Symbol table data structure properties:
- All operations should be very fast (preferably O(1)).
- Must be able to disambiguate overloaded name use (depending on language): add type, scope, nesting info to lookup.
- Must not be affected by typical programmer “clustered” names: x1, x11, x12, etc.

Best bet:
- Use a hash table (or a list or tree or hash table of hash tables).
- Separate chains better than a closed array (chains handled as little stacks, insertions and deletions always at the front).
- Hash function needs to use all characters in a name (to avoid collisions), and involve character position too!

Example:

<table>
<thead>
<tr>
<th>Indices</th>
<th>Buckets</th>
<th>Lists of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>i •</td>
</tr>
<tr>
<td>1</td>
<td>size</td>
<td>3 •</td>
</tr>
<tr>
<td>2</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>temp</td>
<td>•</td>
</tr>
<tr>
<td>4</td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>
Sample hash function code:

```c
#include <stdio.h>

#define SIZE 211     // typically a prime number
#define SHIFT 4

int hash ( char * key )
{
    int temp = 0;
    int i = 0;
    while (key[i] != '\0')
    {
        temp = ((temp << SHIFT) + key[i]) % SIZE;
        ++i;
    }
    return temp;
}
```

Easy way to get O(1) behavior when exiting a scope: use a linked list (or tree or... of hash tables, one hash table for each scope:

```c
class A { void f(); }

void A::f() // go back inside A
{ ... }
```

Two additional scope issues (of many):
- Recursion: insertion into table must occur before processing is complete:
  ```c
  // lookup of f in body must work:
  void f() { ... f() ... }
  ```
- Relaxation of declaration before use rule (C++ and Java class scopes): all insertions must occur before all lookups (two passes required):
  ```c
  class A
  { int f() { return x; } int x; }
  ```

One more scope issue: dynamic scope
- Some languages use a run-time version of scope that does not follow the layout of the program on the page, but the execution path: LISP, perl.
- Symbol table then must be part of runtime system, providing lookup of names during execution (it better be really fast in this case).

Some structure similar to the previous slide is actually required in C++, Ada, and other languages where scopes can be arbitrarily re-entered (C++ has the scope resolution operator `::`), since individual scopes must be attached to names, allowing them to be "called":

```c
class A { void f(); }

void A::f() // go back inside A
{ ... }
```

- Called “dynamic scope” (vs. the more usual lexical or static scope).
- A questionable design choice for any but the most dynamic, interpreted languages, since there can then be no static semantic analysis (no static type checking, for example)
- Running the symbol table during execution also slows down execution speed substantially
Example of dynamic scope (C syntax):

```c
int i = 1;
void f(void)
{ printf("%d\n",i);}

main()
{ int i = 2;
  /* the following call prints 1 using normal lexical
     scoping, but prints 2 (the value of the local i) 
     using dynamic scope */
  f();
  return 0;
}
```

TINY symbol table:

- All names are global: there are no scopes.
- Declaration is by use: if a lookup fails, perform an insert.
- Virtually no information has to be kept (all names are int vars), so I had to invent something to store in the symbol table (line numbers).
- No deletes!

TINY symtab.h:

```c
/* Insert line numbers and memory locs 
into the symbol table */
void st_insert( char * name, int lineno, int loc );

/* return the memory 
location of a variable or -1 if not found */
int st_lookup ( char * name );

/* Procedure printSymTab prints a formatted 
listing of the symbol table contents 
to the listing file */
void printSymTab(FILE * listing);
```

C-Minus Symbol Table

- Use basic structure of TINY
- Store tree pointers
- Add enterScope() and exitScope()
- List of tables structure helpful (slide 15)
- Add nesting level to tree nodes
- Add pointer to declaration in all ID nodes (found by lookup)
- Use best ADT methods (hide all details of actual symtab structure)

C-Minus symtab.h:

```c
/* Start a new scope; return 0 if malloc fails, 
ext else 1 */
int st_enterScope(void);

/* Remove all declarations in the current scope */
void st_exitScope(void);

/* Insert def nodes from the syntax tree 
return 0 if malloc fails, else 1 */
int st_insert( TreePtr );

/* Return the defnode of a variable, parameter, or 
function, or NULL if not found */
TreePtr st_lookup ( char * name );
```
Data types and type checking

- A data type is constructed recursively out of simple or base types (int, char, double, etc.) and type constructors that create "new" types out of a group of existing ones: struct, union, * ("pointer to"), enum, []("array of"), etc.
- Types in code are checked by examining the "compatibility" of the types of the components, and by determining a "result" type, if any, from these.

C Example

- Suppose a function is declared as char * f(double d)
- Data type of f is then char*() (double) (function from double to char*)
- The call f(2) type checks because f is a function, 2 is an int, and int is compatible in C with double (can be silently converted). The result then must be of type char*

In terms of syntax tree:

```
<table>
<thead>
<tr>
<th>call</th>
<th>char*</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: f</td>
<td>num: 2</td>
</tr>
<tr>
<td>char*() (double)</td>
<td>int</td>
</tr>
<tr>
<td>(1) is function</td>
<td>(2) compatible with</td>
</tr>
<tr>
<td>(3) result has type</td>
<td></td>
</tr>
</tbody>
</table>
```

Type compatibility of constructed types

- Generally depends on a notion of when two type are "equal" (equivalent), or at least closely related.
- C example:
  ```c
  struct {} x,z;
  struct {} y;
  y = x; // illegal! (different types)
  z = x; // ok! Same types
  ```

- On the other hand:
  ```c
  struct A {} x;
  struct A y;
  y = x; // now it's ok!
  ```

Type Equivalence Algorithm

- Structural equivalence: as long as the types have the same structure, they are equivalent.
- Name equivalence: types are equivalent only if they are identical as names
- Declaration equivalence: types are equivalent if they lead back (through renaming) to the same original use of a type constructor.
Equivalence Example (C syntax)

- struct A {};
- typedef struct A A;
- typedef struct {} B;
- struct A x; A y; B z;
- x, y, z all structurally equivalent
- x, y declaration equivalent, but z is not declaration equivalent to these
- none are name equivalent

C uses a combination of structural and declaration equivalence:

- Declaration equivalence for struct and union
- Structural equivalence for arrays, pointers, and functions
- enum isn’t even a type constructor, but constructs a named subrange of int (unlike C++ - see next slide)

Digression: Enums in C and C++

- An enum in C is not a real type constructor:
  - enum A {one,two,three} x;
  - enum B {four,five,six} y;
  - x = y /* ok in C */
- In C++ this assignment is an error:
  - C:\classes\cs153\f02\gxx enum.cpp
  - enum.cpp: In function ‘int main()’: enum.cpp:7: cannot convert ‘B’ to ‘A’ in assignment
- Note how error message implies that C++ automatically generates a typedef enum A A!

Digression on C function types

- There are two kinds of function types in C that are almost identical (and that can almost be used interchangeably) - function constants and function pointers:
  - typedef char* F(double);
  - typedef char* (*G)(double);
- F is a “constant” function type (a prototype), while G is a “pointer to function” type, or function variable:
  - F f; // a prototype for a func f
  - G g = f; // g is var init’ed to f
  - f = g; // illegal - f is const

Representing types internally in a compiler

- Since types are built up recursively, a tree structure must be used (syntax tree gets another major node kind: datatype).
- Some languages (FORTRAN, TINY, C-Minus) have flat type spaces, so that an enum can be used: int, intarray, function.
In many ways, this mirrors the close relationship in C between pointers and arrays:

```c
int x[10];
int* y = x; // ok
x = y; // illegal
```

In calls and params it really doesn’t matter which type you use or assume:

```c
f(2), (*f)(2) and (&f)(2) all work fine, and void p(f f) and void p(G gg) are identical in effect.
```

### Recursive types

- Present special problems:
  ```c
  struct A { int x; struct A next; };
  ```
  is illegal, because it would represent an “infinite” type (just as `void f(void) { f(); }`) represents an “infinite” call).

- In C must interpose a pointer:
  ```c
  struct A { int x; struct A* next; };
  ```

- Some languages use a union instead.
- Others (like Java) have implicit pointers.

### Other issues (a sample)

- Should array size be part of its type? (C says no)
- How far should compatibility of types go? (Should any two pointers be compatible?)
- Dynamic typing: constructing types during execution.

---

**Type checking in TINY**

- Only two types: int and bool
- Only need to check if statement, while statement, assignment, and a few other cases
- Type errors may create a “void” type. Suppress error messages in the presence of void.

---

**Sample TINY type checking code**

```c
switch (t->kind.exp)
{ case OpK:
    if (((t->child[0]):-type != Integer) ||
         (t->child[1]):-type != Integer))
         typeError(t,"Op applied to non-integer");
    else
        if (t->attr.op == EQ) || (t->attr.op == LT))
            t->type = Boolean;
        else
            t->type = Integer;
    break;
```

---

**Type Checking in C-Minus**

- Go through Appendix A carefully, writing out all type rules
- As in TINY, there are only a few types (other than functions). And there are no explicit function types, or function variables or parameters. Also no recursive types. And no typedefs.
- Answer questions such as: is `x = y` legal if `x` and `y` are both arrays?
Example from Appendix A

18. expression \(\rightarrow\) \(\mathit{var} =\) expression \| simple-expression

19. \(\mathit{var} \rightarrow \mathit{ID} \mid \mathit{ID} \ [\mathit{expression}]\)

An expression is a variable reference followed by an expression, or just a simple expression. The assignment has the usual storage semantics: the location of the variable represented by \(\mathit{var}\) is found, then the subexpression to the right of the assignment is evaluated, and the value of the subexpression is stored at the given location. This value is also returned as the value of the entire expression. A \(\mathit{var}\) is either a simple (integer) variable or a subscripted array variable. A negative subscript causes the program to halt (unlike C). However, upper bounds of subscripts are not checked.

Making syntax tree traversals easy: use “generic” traversal function:

```c
static void traverse(TreeNode * t, 
        void (* preProc) (TreeNode *), 
        void (* postProc) (TreeNode *) ) 
{ if (t != NULL) 
    { int i; 
      for (i=0; i < MAXCHILDREN; i++) 
          traverse(t->child[i], preProc, postProc); 
    } 
    postProc(t); 
    traverse(t->sibling, preProc, postProc); 
}
```

// builds symtab in preorder:
traverse(syntaxTree,insertNode,nullProc);

// checks types in postorder:
traverse(syntaxTree,nullProc,checkNode);

```c
void nullProc(treeNode* t) 
{ }
```

etc . . .

Analyze.h - a two-step process:

/* Function buildSymtab constructs the symbol * table by preorder traversal of the syntax tree */
void buildSymtab(TreeNode *);

/* Procedure typeCheck performs type checking * by a postorder syntax tree traversal */
void typeCheck(TreeNode *);

What should C-Minus Print under TraceAnalyze?

- Possibly a representation of the symbol table, as in TINY
- But also another representation of the tree with types added
- PrintTree could be modified to do this, or a new PrintTypes function added to util.h/util.c

An Example of C-Minus Symbol Table Construction and the use of the symbol table to link uses of names to their defs.

CS 153 - Fall, 2002 - K. Louden - 11/10/02
The Example:

```c
int a; /*d1*/
int b[10]; /*d2*/
int c /*d3*/ (int a[] /*d4*/, int c /*d5*/)
{ /* Position 1 */
  if (c)
  {
    int d; /*d6*/ /* Position 2 */
    d = a[c] + b[c];
    return d;
  }
  return 0;
}
void main(void) /*d7*/
{ /* Position 3 */
  output(c(b,a));
}
```

Syntax tree:

```
[call
  [a
    [block
      [b
        [c
          [main
            [call output
              [c
                [b
                  [a]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]}
Symbol Table at Position 3:

Lookups of *output*, *a*, *b*, and *c* after pos. 3 produces the following tree with links: