CS415 Compilers

Typing, Symbol Tables, Intermediate Representations, Code Generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Type system: Each language construct (operator, expression, statement, ...) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for
- basic types: integer, char, real, boolean, typeError
- constructed types, e.g., one-dimensional arrays: array(lb, ub, elem_type), where elem_type is a type expression

A type checker implements a type system. It computes or “constructs” type expressions for each language construct.
Programmers may define their own types and give them names:

```plaintext
type my_int is int;
...
int a;
my_int b;
...
... a + b ...
```

*Type names can also be part of the type expression language.*

*Note: type names and type variables are different!*
Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

```pascal
type A is array(1..10) of integer;
type B is array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence:

name equivalence:
Type Equivalence

**Structural** -- type equivalence: type names are expanded

**Name** -- type equivalence: type names are not expanded

Example:

```plaintext
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence:  
(a, b, c, d, e)

name equivalence:  
(a); (b); (c, d, e);
Revisit our type inference rule for “+”.

\[
\text{exp : exp } '+' \text{ exp } \begin{cases} 
\text{if ($1 == \text{integer} \land $3 == \text{integer})} \\
\quad \quad $2 = \text{integer}; \\
\text{else} \\
\quad \quad $2 = \text{typeError}; \\
\quad \quad \text{printf(“
***Error: illegal operand types
”);}
\end{cases}
\]

PROJECT HINT: The definition of type expression as C types (structs) should be done in \texttt{attr.h}. \texttt{attr.c} may contain helper functions. The assignment of type expression C types to terminals and nonterminals of the grammar is done in \texttt{parse.y}.
Lexically-scoped Symbol Tables

The problem
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface
- `insert(name, level)` - creates record for `name` at `level`
- `lookup(name, level)` - returns pointer or index
- `delete(level)` - removes all names declared at `level`

Many implementation schemes have been proposed (see § B.4)
- We’ll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We’ll see the corresponding run-time structures that are used to establish addressability later.
procedure p {
  int a, b, c
  procedure q {
    int v, b, x, w
    procedure r {
      int x, y, z
      ...
    }
    ...
  }
  ...
  r ... s
}
...
q ...

B0: {
  int a, b, c
}
B1: {
  int v, b, x, w
}
B2: {
  int x, y, z
  ...
}
B3: {
  int x, a, v
  ...
}
...

Lecture 18
Example

procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ...
        }
    }
    procedure s {
        int x, a, v
        ...
    }
    ... r ... s
}
...
...
...
...
...
q ...

B0: {
    int a_0, b_1, c_2
    B1: {
        int v_3, b_4, x_5, w_6
        B2: {
            int x_7, y_8, z_9
            ...
        }
        B3: {
            int x_{10}, a_{11}, v_{12}
            ...
        }
    }
    ...
    ...
    no y or z
    a_{11}, b_4, c_2,
    v_{12}, w_6, x_{10},
    no y or z
}
Lexically-scoped Symbol Tables

High-level idea  -- Example: \( p \{ q \{ r, s \} \} \)

- Create a new table for each scope
- Chain them together for lookup

“Chain of tables” implementation

- \textit{insert()} may need to create table
- It always inserts at current level
- \textit{lookup()} walks chain of tables & returns first occurrence of name
- \textit{delete()} throws away table for level \( p \), if it is top table in the chain

Individual tables can be hash tables.
Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup

--- Example: \( p \{ q \{ r, s \} \} \)

\[
\begin{array}{c|c|c}
& q & \hline
s & a & \hline
x & b & \hline
& c & \hline
q & \hline
v & \hline
\end{array}
\]

Remember

- If we add the subscripts, the relationship between the code and the table becomes clear

\( a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, \text{no } y \text{ or } z \)

If we add the subscripts, the relationship between the code and the table becomes clear.

Can also be

\( a_2, b_1, c_0, v_2, w_1, x_1, \text{no } y \text{ or } z \)
Implementing Lexically Scoped Symbol Tables

Stack organization -- Example: \( p \{ q \{ r, s \} \} \)

Implementation

- **insert ()** creates new level pointer if needed and inserts at nextFree
- **lookup ()** searches linearly from nextFree-1 backward
- **delete ()** sets nextFree to the start location of the level deleted.

Advantage
- Uses much less space

Disadvantage
- Lookups can be expensive
Stack organization  -- Example: `p { q { r, s } }`

```
<table>
<thead>
<tr>
<th>Level</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b, c, v</td>
</tr>
<tr>
<td>1</td>
<td>w, x, y, z</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, v</td>
</tr>
<tr>
<td>3</td>
<td>a, b, c, v</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, v</td>
</tr>
</tbody>
</table>
```

Implementation

- **insert()** creates new level pointer if needed and inserts at `nextFree`
- **lookup()** searches linearly from `nextFree-1` backward
- **delete()** sets `nextFree` to the start location of the level deleted.

Advantage
- Uses much less space

Disadvantage
- Lookups can be expensive
Threaded stack organization

Implementation

• **insert ()** puts new entry at the head of the list for the name
• **lookup ()** goes direct to location
• **delete ()** processes each element in level being deleted to remove from head of list

Advantage

• lookup is fast

Disadvantage

• delete takes time proportional to number of declared variables in level
Threaded stack organization

- Implementation
  - `insert()` puts new entry at the head of the list for the name
  - `lookup()` goes direct to location
  - `delete()` processes each element in level being deleted to remove from head of list

**Advantage**
- `lookup` is fast

**Disadvantage**
- `delete` takes time proportional to number of declared variables in level

Diagram:
- Variables: `a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z`
- `h(x)`
- `growth`
- `a: 12`, `v: 11`, `x: 10`, `w: 6`, `x: 5`, `b: 4`, `v: 3`, `c: 2`, `b: 1`, `a: 0`
Intermediate Representations
(EaC Chapter 5)

- Front end - produces an intermediate representation (IR)
- Middle end - transforms the IR into an equivalent IR that runs more efficiently
- Back end - transforms the IR into native code

- IR encodes the compiler’s knowledge of the program
- Middle end usually consists of several passes
Intermediate Representations

• Decisions in IR design affect the speed and efficiency of the compiler

• Some important IR properties
  → Ease of generation
  → Ease of manipulation
  → Size
  → Level of abstraction

• The importance of different properties varies between compilers
  → Selecting an appropriate IR for a compiler is critical
Three major categories

• **Structural**
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

• **Linear**
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

• **Hybrid**
  - Combination of graphs and linear code

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Two different representations of an array reference:

- **High level AST:**
  - Good for memory disambiguation

  ![Diagram](image)

- **Low level linear code:**
  - Good for address calculation

```plaintext
loadI 1     => r1
sub  r_j, r1 => r2
loadI 10    => r3
mul  r2, r3 => r4
sub  r_i, r1 => r5
add  r4, r5 => r6
loadI @A   => r7
Add  r7, r6 => r8
load  r8    => r_{Aij}
```
• Structural IRs are usually considered high-level
• Linear IRs are usually considered low-level
• Not necessarily true:

Low level AST

load

loadArray A,i,j

High level linear code

```plaintext
load
+
+
*
-
-

10
i
1

j
1
```
An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed.

- Can use linearized form of the tree
  - Easier to manipulate than pointers
    - \( x - 2 * y \) in postfix form
    - \(- * y 2 x\) in prefix form

- S-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)
A directed acyclic graph (DAG) is an AST with a unique node for each value.

- Makes sharing explicit
- Encodes redundancy

```
z ← x - 2 * y
w ← x / 2
```

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

Originally used for stack-based computers, now Java

• Example:
  \[ x - 2 \times y \]
  becomes
  
  ```
push x
push 2
push y
multiply
subtract
  ```

Advantages

• Compact form
• Introduced names are *implicit*, not *explicit*
• Simple to generate and execute code

Useful where code is transmitted over slow communication links *(the net)*

Implicit names take up no space, where explicit ones do!
Several different representations of three address code

- In general, three address code has statements of the form:
  \[ x \leftarrow y \text{ op } z \]

  With 1 operator (\textit{op}) and, at most, 3 names (\(x, y, z\))

Example:

\[
\begin{align*}
  z &\leftarrow x - 2 \ast y \\
\end{align*}
\]

becomes

\[
\begin{align*}
  t &\leftarrow 2 \ast y \\
  z &\leftarrow x - t \\
\end{align*}
\]

Advantages:
- Resembles many machines
- Introduces a new set of names
- Compact form
Three Address Code: Quadruples

Naive representation of three address code

• Table of $k \times 4$ small integers
• Simple record structure
• Easy to reorder
• Explicit names

RISC assembly code

<table>
<thead>
<tr>
<th>load</th>
<th>r1, y</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>r2, 2</td>
</tr>
<tr>
<td>mult</td>
<td>r3, r2, r1</td>
</tr>
<tr>
<td>load</td>
<td>r4, x</td>
</tr>
<tr>
<td>sub</td>
<td>r5, r4, r3</td>
</tr>
</tbody>
</table>

Quadruples

<table>
<thead>
<tr>
<th>load</th>
<th>1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

The original FORTRAN compiler used “quads”
Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
</tr>
</tbody>
</table>

Implicit names take no space!
Control-flow Graph (CFG)

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  
  → Can be represented with quads or any other linear representation

- Edges in the graph represent control flow

Example

```
if (x = y)
a ← 2
b ← 5
a ← 3
b ← 4
c ← a * b
```

Basic blocks — Maximal length sequences of straight-line code
Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce $\phi$-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
  x ← x + 1
  y ← y + x
```

SSA-form

```
x_0 ← ...
y_0 ← ...
if (x_0 > k) goto next
loop:
x_1 ← φ(x_0, x_2)
y_1 ← φ(y_0, y_2)
x_2 ← x_1 + 1
y_2 ← y_1 + x_2
if (x_2 < k) goto loop
next: ...
```

Strengths of SSA-form

- Sharper analysis
- $\phi$-functions give hints about placement
- (sometimes) faster algorithms
Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce $\phi$-functions to make it work

**Original**

```
x ← ...
y ← ...
while (x < k)
x ← x + 1
y ← y + x
```

**SSA-form**

```
x_0 ← ...
y_0 ← ...
if (x_0 > k) goto next
loop:
x_1 ← \phi(x_0,x_2)
y_1 ← \phi(y_0,y_2)
x_2 ← x_1 + 1
y_2 ← y_1 + x_2
if (x_2 < k) goto loop
next: ...```

**Strengths of SSA-form**

- Sharper analysis
- $\phi$-functions give hints about placement
- (sometimes) faster algorithms
Code generation
Read EaC: Chapter 7.1 - 7.5