Last Week: Types

- What is a type?
  - Simplest answer: a set of values
  - Integers, real numbers, booleans, ...

- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
  - Hide implementation details
  - Documentation
  - Optimization

Last Week: Type System

- A type system of a programming language is a way to define how "good" program behave
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime

- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)
Strongly vs. weakly typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

(YMMV, not everybody agrees on this classification)

```perl
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

```java
main() {
    int a = 31;
    char b[3] = "42x";
    int c = a+b;
}
```

Last week: how does this magic happen?

- We probably need to go over the AST?
- how does this relate to the clean formalism of the parser?

Syntax Directed Translation

- The parse tree (syntax) is used to drive the translation
- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - How to update the attributes when a production is used in a derivation
- Attribute grammars

Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      ```
      Expr -> Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise
      ```
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions
- \( \text{Expr} \rightarrow \text{Expr} + \text{Term} \)
  Becomes
  \( \text{Expr} \rightarrow \text{Expr}_1 + \text{Term} \)

Example

Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

Dependencies

- A semantic equation \( a = b_1, \ldots, b_m \) requires computation of \( b_1, \ldots, b_m \) to determine the value of \( a \)
  - The value of \( a \) depends on \( b_1, \ldots, b_m \)
    - We write \( a \leftarrow b_i \)
Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values

![Diagram of AST and Dependence graph]

Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering
- Works as long as there are no cycles

Building Dependency Graph

- All semantic equations take the form
  
  \[ \text{attr}_1 = \text{func}_1(\text{attr}_{1.1}, \text{attr}_{1.2}, \ldots) \]
  
  \[ \text{attr}_2 = \text{func}_2(\text{attr}_{2.1}, \text{attr}_{2.2}, \ldots) \]

- Actions with side effects use a dummy attribute
- Build a directed dependency graph \( G \)
  
  - For every attribute \( a \) of a node \( n \) in the AST create a node \( n.a \)
  
- For every node \( n \) in the AST and a semantic action of the form \( b = f(c_1, c_2, \ldots, c_k) \) add edges of the form \( (c_i, b) \)

Example

![Example diagram with semantic rules and actions]
Example

### Topological Order

- For a graph $G=(V,E)$, $|V|=k$
- Ordering of the nodes $v_1,v_2,...,v_k$ such that for every edge $(v_i,v_j) \in E$, $i < j$

Example topological orderings: $1\ 4\ 3\ 2\ 5$, $4\ 1\ 3\ 5\ 2$

- But what about cycles?
  - For a given attribute grammar hard to detect if it has cyclic dependencies
    - Exponential cost
  - Special classes of attribute grammars
    - Our “usual trick”
    - Sacrifice generality for predictable performance
Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node
- Attributes of tokens are technically considered as synthesized attributes

**S-attributed Grammars**

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes
- Can be computed by any bottom-up parser **during parsing**
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children
Example: typesetting

- Vertical geometry
  - fontsize (ps) – size of letters in a box. Subscript text has smaller point size of 0.7p.
  - baseline
  - height (ht) – distance from top of the box to the baseline
  - depth (dp) – distance from baseline to the bottom of the box.

L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X_1...X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ that only depends on
    - Attributes of $X_1...X_{j-1}$ to the left of $X_j$, or
    - Inherited attributes of $A$

Example: typesetting

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow B$</td>
<td>$B.ps = 10$</td>
</tr>
<tr>
<td>$B \rightarrow B_1 B_2$</td>
<td>$B_1.ps = B.ps$</td>
</tr>
<tr>
<td></td>
<td>$B_2.ps = B.ps$</td>
</tr>
<tr>
<td></td>
<td>$B.ht = max(B_1.ht, B_2.ht)$</td>
</tr>
<tr>
<td></td>
<td>$B.dp = max(B_1.dp, B_2.dp)$</td>
</tr>
<tr>
<td>$B \rightarrow B_1_{sub} B_2$</td>
<td>$B_1.ps = B.ps$</td>
</tr>
<tr>
<td></td>
<td>$B_2.ps = 0.7B.ps$</td>
</tr>
<tr>
<td></td>
<td>$B.ht = max(B_1.ht, B_2.ht - 0.25*B.ps)$</td>
</tr>
<tr>
<td></td>
<td>$B.dp = max(B_1.dp, B_2.dp - 0.25*B.ps)$</td>
</tr>
<tr>
<td>$B \rightarrow text$</td>
<td>$B.ht = getHt(B.ps, text.lexval)$</td>
</tr>
<tr>
<td></td>
<td>$B.dp = getDp(B.ps, text.lexval)$</td>
</tr>
</tbody>
</table>
Attribute grammars: summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not "local"
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: S-attributed, L-attributed

Intermediate Representation

- "neutral" representation between the front-end and the back-end
  - Abstracts away details of the source language
  - Abstracts away details of the target language
- A compiler may have multiple intermediate representations and move between them
  - In practice, the IR may be biased toward a certain language (e.g., GENERIC in gcc)

Intermediate Representation(s)

- Annotated abstract syntax tree
- Three address code
- ...

Example: Annotated AST

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id = E</td>
<td>S.nptr = makeNode(&quot;assign&quot;, makeLeaf(id.id.place), E.nptr)</td>
</tr>
<tr>
<td>E → E1 + E2</td>
<td>E.nptr = makeNode(&quot;+&quot;, E1.nptr, E2.nptr)</td>
</tr>
<tr>
<td>E → E1 * E2</td>
<td>E.nptr = makeNode(&quot;*&quot;, E1.nptr, E2.nptr)</td>
</tr>
<tr>
<td>E → -E1</td>
<td>E.nptr = makeNode(&quot;uminus&quot;, E1.nptr)</td>
</tr>
<tr>
<td>E → (E1)</td>
<td>E.nptr = E1.nptr</td>
</tr>
<tr>
<td>E → id</td>
<td>E.nptr = makeLeaf(id.id.place)</td>
</tr>
</tbody>
</table>

- makeNode – creates new node for unary/binary operator
- makeLeaf – creates a leaf
- id.place – pointer to symbol table
Example

Three Address Code (3AC)

- Every instruction operates on three addresses
  - result = operands operator operand2
- Close to low-level operations in the machine language
  - Operator is a basic operation
- Statements in the source language may be mapped to multiple instructions in three address code

Three address code: example

Three address code: example instructions
Array operations

- Are these 3AC operations?

```plaintext
x := y[i]
```

```plaintext
\begin{align*}
t1 & := y \quad ; \quad t1 = \text{address of } y \\
t2 & := t1 + i \quad ; \quad t2 = \text{address of } y[i] \\
X & := *t2 \quad ; \quad \text{value stored at } y[i]
\end{align*}
```

```plaintext
x[i] := y
```

```plaintext
\begin{align*}
t1 & := x \quad ; \quad t1 = \text{address of } x \\
t2 & := t1 + i \quad ; \quad t2 = \text{address of } x[i] \\
*t2 & := y \quad ; \quad \text{store through pointer}
\end{align*}
```

Three address code: example

```plaintext
int main(void) {
    int i;
    int b[10];
    for (i = 0; i < 10; ++i)
        b[i] = i*i;
}
```

```plaintext
\begin{align*}
i & := 0 \quad ; \quad \text{assignment} \\
L1: & \text{if } i \geq 10 \text{ goto } L2 \quad ; \quad \text{conditional jump} \\
t0 & := i*i \quad ; \quad \text{address-of operation} \\
t1 & := &b \quad ; \quad \text{address of } b[i] \\
t2 & := t1 + i \quad ; \quad \text{t2 holds the address of } b[i] \\
*t2 & := t0 \quad ; \quad \text{store through pointer} \\
i & := i + 1 \\
goto L1
\end{align*}
```

Three address code

- Choice of instructions and operators affects code generation and optimization
- Small set of instructions
  - Easy to generate machine code
  - Harder to optimize
- Large set of instructions
  - Harder to generate machine code
- Typically prefer small set and smart optimizer

Creating 3AC

- Assume bottom up parser
  - Why?
- Creating 3AC via syntax directed translation
- Attributes
  - code – code generated for a nonterminal
  - var – name of variable that stores result of nonterminal
  - freshVar – helper function that returns the name of a fresh variable
Creating 3AC: expressions

 production | semantic rule
---|---
$S \rightarrow \text{id} := E$ | $S.code := E.code || \text{gen}(\text{id}.var := E.var)$
$E \rightarrow E_1 + E_2$ | $E.var := \text{freshVar}();$
$E.code := E_1.code || E_2.code || \text{gen}(E.var := E_1.var + E_2.var)$
$E \rightarrow E_1 * E_2$ | $E.var := \text{freshVar}();$
$E.code := E_1.code || E_2.code || \text{gen}(E.var := E_1.var * E_2.var)$
$E \rightarrow -$ | $E.var := \text{freshVar}();$
$E.code := E_1.code || \text{gen}(E.var := \text{uminus} E_1.var)$
$E \rightarrow (E_1)$ | $E.var := E_1.var$
$E.code := '(' || E_1.code || ')$
$E \rightarrow \text{id}$ | $E.var := \text{id}.var;$
$E.code := ''$

(we use $||$ to denote concatenation of intermediate code fragments)

Creating 3AC: control statements

- 3AC only supports conditional/unconditional jumps
- Add labels
- Attributes
  - begin – label marks beginning of code
  - after – label marks end of code
- Helper function freshLabel() allocates a new fresh label

Creating 3AC: control statements

 production | semantic rule
---|---
$S \rightarrow \text{while } E \text{ do } S_1$ | $S.begin := \text{freshLabel}();$
$S.after := \text{freshLabel}();$
$S.code := \text{gen}('S.begin') || E.code ||$
$\text{gen}('if' E.var = '0' 'goto' S.after) || S_1.code || \text{gen}('goto' S.begin) || \text{gen}('S.after')$
Representing 3AC

- Quadruple (op, arg1, arg2, result)
- Result of every instruction is written into a new temporary variable
- Generates many variable names
- Can move code fragments without complicated renaming
-Alternative representations may be more compact

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 = - c</td>
<td>c</td>
<td></td>
<td>t1</td>
</tr>
<tr>
<td>t2 = b * t1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3 = - c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4 = b * t3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5 = t2 * t4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = t5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allocating Memory

- Type checking helped us guarantee correctness
- Also tells us
  - How much memory allocate on the heap/stack for variables
  - Where to find variables (based on offsets)
  - Compute address of an element inside array (size of stride based on type of element)

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P -&gt; D</td>
<td>{ offset := 0 }</td>
</tr>
<tr>
<td>D -&gt; D</td>
<td></td>
</tr>
<tr>
<td>D -&gt; T D</td>
<td>enter(id.name, T.type, offset) offset := T.width</td>
</tr>
<tr>
<td>T -&gt; Integer</td>
<td></td>
</tr>
<tr>
<td>T -&gt; float</td>
<td>[ T.type = float; T.width = 8 ]</td>
</tr>
<tr>
<td>T -&gt; T[num]</td>
<td>T.type = array(num.val, T.type); T.width = num.val * T.type.width</td>
</tr>
<tr>
<td>T -&gt; *T1</td>
<td>T.type = pointer(T1.type); T.width = 4</td>
</tr>
</tbody>
</table>

Allocating Memory

- Global variable "offset" with memory allocated so far

produce | semantic rule |
---------|---------------|
F -> D   | { offset := 0 } |
D -> D   |               |
D -> T D | enter(id.name, T.type, offset) offset := T.width |
T -> Integer | 
T -> float | [ T.type = float; T.width = 8 ] |
T -> T[num] | T.type = array(num.val, T.type); T.width = num.val * T.type.width |
T -> *T1 | T.type = pointer(T1.type); T.width = 4 |
Adjusting to bottom-up

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → MD</td>
<td></td>
</tr>
<tr>
<td>M → g</td>
<td>offset := 0</td>
</tr>
<tr>
<td>D → DD</td>
<td></td>
</tr>
<tr>
<td>D → T D</td>
<td></td>
</tr>
<tr>
<td>T → integer</td>
<td>T.type := int; T.width := 4</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type := float; T.width := 8</td>
</tr>
<tr>
<td>T → T[i]</td>
<td>T.type := array(num.val, T1.type); T.width := num.val * T1.width;</td>
</tr>
<tr>
<td>T → *T1</td>
<td>T.type := pointer(T1.type); T.width := 4</td>
</tr>
</tbody>
</table>

Generating IR code

- Option 1
  accumulate code in AST attributes

- Option 2
  emit IR code to a file during compilation
    - If for every production the code of the left-hand-side is constructed from a concatenation of the code of the RHS in some fixed order

Expressions and assignments

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>{ p := lookup(id.name); if p ≠ null then emit(p := E.var) else error }</td>
</tr>
<tr>
<td>E → E1 op E2</td>
<td>{ E.var := freshVar(); emit(E.var := E1.var op E2.var) }</td>
</tr>
<tr>
<td>E → not E1</td>
<td>{ E.var := freshVar(); emit(E.var := ~E1.var) }</td>
</tr>
<tr>
<td>E → (E)</td>
<td>{ E.var := E.var }</td>
</tr>
<tr>
<td>E → true</td>
<td>{ E.var := freshVar(); emit(E.var := 't') }</td>
</tr>
<tr>
<td>E → false</td>
<td>{ E.var := freshVar(); emit(E.var := 'f') }</td>
</tr>
</tbody>
</table>

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false

Boolean Expressions
Boolean expressions via jumps

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| E → id1 op id2 | \[
E.var := freshVar();
emit('if' id1.var relop id2.var 'goto' nextStmt+2);
emit(E.var := "V");
emit('goto' nextStmt + 1);
emit(E.var := "X")
\] |

Example

Short circuit evaluation

- Second argument of a boolean operator is only evaluated if the first argument does not already determine the outcome
- (x and y) is equivalent to if x then y else false;
- (x or y) is equivalent to if x then true else y

Example

\[
\begin{align*}
&if \ a < b \ goto \ 103 \\
&if \ c < d \ goto \ 107 \\
&if \ e < f \ goto \ 111 \\
&if \ !(c < d) \ goto \ 103 \\
\end{align*}
\]
More examples

```c
int denom = 0;
if (denom == 0) {
    oops_i_just_divided_by_zero();
}
```

```c
int x=0;
if (++x>0 && x==1) {
    hmmm();
}
```

Summary
- Three address code (3AC)
- Generating 3AC
- Boolean expressions
- Short circuit evaluation

Next time
- Generating IR for control structures
  - While, for, if
- backpatching

The End