Last Week: Attribute Grammars

- Adding attributes + actions to a grammar
- Evaluating attributes
  - Build AST
  - Build dependency graph
  - Evaluation based on topological order
  - (works as long as there are no cycles)
- L-attributes, S-attributed grammars
  - Pre-determined evaluation order
  - Can be integrated into parsing

Last Week: 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
  - can be represented as "quads"
    (result, operand1, operator, operand2)
Last Week: Creating 3AC

- Assume bottom up parser
  - Covers a wider range of grammars
  - LALR sufficient to cover most programming languages
- 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

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E$</td>
<td>$S.\text{code} = E.\text{code} \</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 + E_2$ | $E.\text{var} = \text{freshVar()};$

$E.\text{code} = E_1.\text{code} \ || \ E_2.\text{code} \ || \ \text{gen(E.\text{var} = \text{'}*E_1.\text{var} + E_2.\text{var}\text{'})}$ |
| $E \rightarrow E_1 * E_2$ | $E.\text{var} = \text{freshVar()};$

$E.\text{code} = E_1.\text{code} \ || \ E_2.\text{code} \ || \ \text{gen(E.\text{var} = \text{'}*E_1.\text{var} * E_2.\text{var}\text{'})}$ |
| $E \rightarrow (E_1)$ | $E.\text{var} = \text{freshVar()};$

$E.\text{code} = \text{'}*\text{'}E_1.\text{code}\text{'}*\text{'}$ |
| $E \rightarrow \text{id}$ | $E.\text{var} = \text{id.var};$ $E.\text{code} = \text{'}*\text{'}$ |

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

example

Three address code: example

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

```c
i := 0 ; assignment
L1: if i >= 10 goto L2 ; conditional jump
t0 := i+1 ;
t1 := 4b ; address of operation
t2 := t1 + i ;
*t2 := t0 ; store through pointer
i := i + 1
goto L1
L2:
```

(example source: wikipedia)
Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once

```
p = a + b
q = p - c
p = q + d
q = p * q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
q2 = p2 + q2
```

SSA

- \( \phi \) (phi) function combines different definitions
- \( \phi \) returns the value of \( x_1 \) if control passes through the true branch and the value of \( x_2 \) if it passed through the false branch

```
if (f)
x = 42;
else
x = 73;
y = x * a;
```

```
if (f)
x1 = 42;
else
x2 = 73;
x3 = \( \phi \)(x1,x2);
y = x3 * a;
```

SSA why should we care?

- makes it easy to apply many optimizations
  - constant propagation, dead code elimination...

```
x = 42
x = 73
y = x
```

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

\[ S \rightarrow \text{while } E \text{ do } S_1 \]

### 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 \rightarrow D )</td>
<td>( \text{offset} := 0 )</td>
</tr>
<tr>
<td>( D := D \ D )</td>
<td>( )</td>
</tr>
<tr>
<td>( D := \text{id} \ \text{id} )</td>
<td>( \text{enter(id.name, T.type, offset) offset := T..width) )</td>
</tr>
<tr>
<td>( T := \text{Integer} )</td>
<td>( T.type := \text{int}, T..width = 4 )</td>
</tr>
<tr>
<td>( T := \text{Float} )</td>
<td>( T.type := \text{float}, T..width = 8 )</td>
</tr>
<tr>
<td>( T := T\text{[num]} )</td>
<td>( T.type := \text{array}(num..val, T1..type), T..width = num..val * T1..width )</td>
</tr>
<tr>
<td>( T := *T1 )</td>
<td>( T.type := \text{pointer}(T1..type), T..width = 4 )</td>
</tr>
</tbody>
</table>
Adjusting to bottom-up

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → M D</td>
<td>[ offset := 0]</td>
</tr>
<tr>
<td>M → g</td>
<td></td>
</tr>
<tr>
<td>D → T D</td>
<td>[ enter(id.name, T.type, offset); offset += T.width ]</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 → T1[num]</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 → ~E1</td>
<td>[ E.var := freshVar(); emit(E.var := ~E1.var) ]</td>
</tr>
<tr>
<td>E → true</td>
<td>[ E.var := freshVar(); emit(E.var := ’1’) ]</td>
</tr>
<tr>
<td>E → false</td>
<td>[ E.var := freshVar(); emit(E.var := ’0’) ]</td>
</tr>
</tbody>
</table>

Boolean Expressions

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false
Boolean expressions via jumps

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>
| E → id1 op id2 | \[
\begin{align*}
E.\text{var} & \leftarrow \text{freshVar}(); \\
\text{emit}(&E.\text{var}\text{ relop id2.}\text{var}'\text{goto' nextStmt}+2); \\
\text{emit}(&E.\text{var}\leftarrow'0'); \\
\text{emit}(&E.\text{var}\leftarrow'1'); \\
\text{emit}(&E.\text{var}\leftarrow'x') \\
\end{align*}
\] |

Example

Short circuit evaluation

- Second argument of a Boolean operator is only evaluated if the first argument does not already determine the outcome.

- \((x \text{ and } y)\) is equivalent to \(\text{if } x \text{ then } y \text{ else false;}\)

- \((x \text{ or } y)\) is equivalent to \(\text{if } x \text{ then true else } y\)

Example

<table>
<thead>
<tr>
<th>naive</th>
<th>Short circuit evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: if (a &lt; b) goto 103</td>
<td>101: if (c &lt; d) goto 105</td>
</tr>
<tr>
<td>102: goto 104</td>
<td>103: (T_1 := 0)</td>
</tr>
<tr>
<td>104: if (c &lt; d) goto 107</td>
<td>105: (T_1 := 0)</td>
</tr>
<tr>
<td>105: (T_1 := 0)</td>
<td>106: goto 108</td>
</tr>
<tr>
<td>107: (T_1 := 1)</td>
<td>108: if (e &lt; f) goto 111</td>
</tr>
<tr>
<td>108: if (e &lt; f) goto 111</td>
<td>109: (T_1 := 0)</td>
</tr>
<tr>
<td>109: (T_1 := 0)</td>
<td>110: goto 112</td>
</tr>
<tr>
<td>110: goto 112</td>
<td>111: (T_1 := 1)</td>
</tr>
<tr>
<td>111: (T_1 := 1)</td>
<td>112: (T_1 := T_2 \text{ and } T_3)</td>
</tr>
<tr>
<td>112: (T_1 := T_2 \text{ and } T_3)</td>
<td>113: (T_1 := T_2 \text{ and } T_3)</td>
</tr>
</tbody>
</table>

\(a < b \text{ or } (c < d \text{ and } e < f)\)
Control Structures

More examples:

```c
int denom = 0;
if (denom == non/denom) {
    oops_i just_divided_by_zero();
}
```

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

Control Structures: next

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → S</td>
<td>S.next = freshLabel(); P.code = S.code</td>
</tr>
<tr>
<td>S → S1S2</td>
<td>S1.next = freshLabel(); S2.next = S.next; S.code = S1.code</td>
</tr>
</tbody>
</table>

- Is S.next inherited or synthesized?
- Is S.code inherited or synthesized?
- The label S.next is symbolic, we will only determine its value after we finish deriving S

Control Structures: conditional

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → if B then S1</td>
<td>B.trueLabel = freshLabel(); B.falseLabel = S.next; S1.next = S.next; S.code = B.code</td>
</tr>
</tbody>
</table>

- Are S1.next, B.falseLabel inherited or synthesized?
- Is S.code inherited or synthesized?
Control Structures: conditional

\[ S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2 \]

- \( B \text{trueLabel} \) and \( B \text{falseLabel} \) considered inherited

Boolean expressions

\[ B \rightarrow \text{B or B} \]

\[ B \rightarrow \text{B and B} \]

\[ B \rightarrow \text{not B} \]

\[ B \rightarrow \text{id1 relop id2} \]

\[ B \rightarrow \text{true} \]

\[ B \rightarrow \text{false} \]

- How can we determine the address of \( B \text{falseLabel} \)?

- Only possible after we know the code of \( B \) and all the code preceding \( B \)
Example

Computing labels

- We can compute the values for the labels but it would require more than one pass on the AST
- Can we do it in a single pass?

Backpatching

- Goal: generate code in a single pass
- Generate code as we did before, but manage labels differently
- Keep labels symbolic until values are known, and then back-patch them
- New synthesized attributes for B
  - B.truelist – list of jump instructions that eventually get the label where B goes when B is true.
  - B.falselist – list of jump instructions that eventually get the label where B goes when B is false.

Backpatching

- For every label, maintain a list of instructions that jump to this label
- When the address of the label is known, go over the list and update the address of the label
- Previous solutions do not guarantee a single pass
  - The attribute grammar we had before is not S-attributed (e.g., next), and is not L-attributed.
Backpatching

- `makelist(addr)` – create a list of instructions containing `addr`
- `merge(p1,p2)` – concatenate the lists pointed to by `p1` and `p2`, returns a pointer to the new list
- `backpatch(p,addr)` – inserts `i` as the target label for each of the instructions in the list pointed to by `p`

Backpatching Boolean expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>B → B1 or M B2</code></td>
<td><code>backpatch(B1.falseList,M.instr); B.trueList = merge(B1.trueList,B2.trueList);</code></td>
</tr>
<tr>
<td><code>B → B1 and M B2</code></td>
<td><code>backpatch(B1.trueList,M.instr); B.falseList = merge(B1.falseList,B2.falseList);</code></td>
</tr>
<tr>
<td><code>B → not B</code></td>
<td><code>B.trueList = B1.falseList; B.falseList = B1.trueList;</code></td>
</tr>
<tr>
<td><code>M → ε</code></td>
<td><code>M.instr = nextInstr;</code></td>
</tr>
</tbody>
</table>

Example

```
X < 100 or x > 200 and x != y
```

Marker

- `{ M.instr = nextInstr; }`
- Use `M` to obtain the address just before `B2` code starts being generated
Example

$X < 100 \text{ or } x > 200 \text{ and } x \neq y$

$B \rightarrow id1 \ \text{rellop} \ id2$
$B.\text{trueList} = \text{makeList}(\text{nextInstr})$;
$B.\text{falseList} = \text{makeList}(\text{nextInstr}+1)$;
$\text{emit}('if' \ id1.\text{var} \ \text{rellop} \ id2.\text{var} ' \text{goto } _') \lor \text{emit}('\text{goto } _')$

$100: \text{if} \ x < 100 \text{ goto } _$
$101: \text{goto } _$
$102: \text{if} \ x > 200 \text{ goto } _$
$103: \text{goto } _$
$104: \text{if} \ x \neq y \text{ goto } _$
**Example**

\[ X < 100 \text{ or } x > 200 \text{ and } x \neq y \]

### Before backpatching

1.00: if x<100 goto _
1.01: goto _
1.02: if x>200 goto 104
1.03: goto _
1.04: if x=y goto _
1.05: goto _

### After backpatching by the production B → B1 and M.B2

Example

**Backpatching for statements**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → (B) M S</td>
<td>backpatch(B.trueList,M instr); S.nextList = merge(S.falseList, S.nextList);</td>
</tr>
<tr>
<td>S → if (B) M S</td>
<td>backpatch(B.trueList, M instr); S.nextList = merge(S.falseList, S.nextList);</td>
</tr>
<tr>
<td>N else M S</td>
<td>\text{emit('goto M instr');}</td>
</tr>
<tr>
<td>S → while M S</td>
<td>\text{emit('goto M instr');}</td>
</tr>
<tr>
<td>M → S</td>
<td>\text{emit('goto M instr');}</td>
</tr>
<tr>
<td>N → M</td>
<td>\text{emit('goto M instr');}</td>
</tr>
<tr>
<td>L → M S</td>
<td>\text{emit('goto M instr');}</td>
</tr>
<tr>
<td>L → S</td>
<td>\text{emit('goto M instr');}</td>
</tr>
</tbody>
</table>
Procedures

- we will see handling of procedure calls in much more detail later

Procedures

\[
\begin{align*}
\text{D} & \rightarrow \text{define } \text{id} \ (F) \ [\ S ] \\
F & \rightarrow \epsilon \ | \ \text{id}, \ F \\
S & \rightarrow \text{return } E; \ | \ .. \\
E & \rightarrow \text{id} \ (A) \ | \ .. \\
A & \rightarrow \epsilon \ | \ E, \ A
\end{align*}
\]

- type checking
  - function type: return type, type of formal parameters
  - within an expression function treated like any other operator

- symbol table
  - parameter names

Summary

- pick an intermediate representation
- translate expressions
- use a symbol table to implement declarations
- generate jumping code for boolean expressions
  - value of the expression is implicit in the control location
- backpatching
  - a technique for generating code for boolean expressions and statements in one pass
  - idea: maintain lists of incomplete jumps, where all jumps in a list have the same target. When the target becomes known, all instructions on its list are "filled in".

Recap

- Lexical analysis
  - regular expressions identify tokens ("words")
- Syntax analysis
  - context-free grammars identify the structure of the program ("sentences")
- Contextual (semantic) analysis
  - type checking defined via typing judgements
  - can be encoded via attribute grammars
- Syntax directed translation
  - attribute grammars
- Intermediate representation
  - many possible IRs
  - generation of intermediate representation
Journey inside a compiler

position = initial + rate * 60

Token Stream

<id,1> <=> <id,2> <+> <id,3> <*> <60>

Journey inside a compiler

symbol | type | data
---|---|---
positional | float | ... |
*initial* | float | ... |
rate | float | ... |

Journey inside a compiler

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Intermediate Representation
Journey inside a compiler

Intermediate Representation

\[
t_1 = \text{intToFloat}(60) \\
t_2 = id3 \times t_1 \\
t_3 = id2 + t_2 \\
id1 = t_3
\]

Optimized

\[
t_1 = id3 \times 60.0 \\
t_2 = t_1 \\
id1 = id2 + t_1
\]

Next time

- Runtime Environments

The End