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 judgments
  - can be encoded via attribute grammars
- Syntax directed translation (SDT)
  - attribute grammars
- Intermediate representation
  - many possible IRs
  - generation of intermediate representation
  - 3AC

Journey inside a compiler

Token Stream

symbol table

AST

Problem 3.8 from [Appel]

A simple left-recursive grammar:

\[ E \rightarrow E + id \]
\[ E \rightarrow id \]

A simple right-recursive grammar accepting the same language:

\[ E \rightarrow id + E \]
\[ E \rightarrow id \]

Which has better behavior for shift-reduce parsing?

Answer

The stack never has more than three items on it. In general, with LR-parsing of left-recursive grammars, an input string of length \(O(n)\) requires only \(O(1)\) space on the stack.

Journey inside a compiler

The stack grows as large as the input string. In general, with LR-parsing of right-recursive grammars, an input string of length \(O(n)\) requires \(O(n)\) space on the stack.
Journey inside a compiler


Production | Semantic Analysis
---|---
S → id | gen(id.var := E.var)
E op E | gen(E.var := E1.var "op" E2.var)
E inttofloat(num) | gen(E.var := inttofloat(num))
E id | E.var := id.var;

Optimized

Code Gen

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADD F R1, R1, R2
ST F id1, R1

You are here

Compile

Source text


Optimized

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
IR So Far...

- many possible intermediate representations
- 3-address code (3AC)
- Every instruction operates on at most three addresses
  - result = operand1 operator operand2
- gets us closer to code generation
- enables machine-independent optimizations
- how do we generate 3AC?

Last Time: Creating 3AC

- 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>$S \rightarrow id := E$</td>
<td>$S.code \rightarrow E.code</td>
</tr>
</tbody>
</table>
| $E \rightarrow E1 + E2$ | $E1.var \rightarrow freshVar();$
| | $E1.code \rightarrow E1.code || E2.code \rightarrow E2.code || gen(E1.var := E1.var + E2.var)$ |
| $E \rightarrow E1 * E2$ | $E1.var \rightarrow freshVar();$
| | $E1.code \rightarrow E1.code || E2.code \rightarrow E2.code || gen(E1.var := E1.var * E2.var)$ |
| $E \rightarrow E1 -$ | $E1.var \rightarrow freshVar();$
| | $E1.code \rightarrow E1.code || gen(E1.var := 'uminus' E1.var)$ |
| $E \rightarrow (E1)$ | $E1.var \rightarrow E1.var$
| | $E1.code \rightarrow '(' || E1.code || ')'$ |
| $E \rightarrow id$ | $E.var \rightarrow id.var; E.code \rightarrow ''$ |

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

Example
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

Expressions and assignments

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → id = E</td>
<td>p = lookup(id.name); if p ≠ null then emit(p ':=' E.var) else error</td>
</tr>
<tr>
<td>E → Exp op E</td>
<td>E.var = freshVar(); emit(E.var ':=' E1.var op E2.var)</td>
</tr>
<tr>
<td>E → E1</td>
<td>E.var = freshVar(); if E.var = 'uminus' then E.var = E1.var</td>
</tr>
<tr>
<td>E → id</td>
<td>p = lookup(id.name); if p ≠ null then E.var ':=' else error</td>
</tr>
</tbody>
</table>

Boolean Expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<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 ':=' 'not' E1.var)</td>
</tr>
<tr>
<td>E → (E)</td>
<td>E.var = freshVar(); emit(E.var ':=' '1')</td>
</tr>
<tr>
<td>E → true</td>
<td>E.var = freshVar(); emit(E.var ':=' '0')</td>
</tr>
<tr>
<td>E → false</td>
<td>E.var = freshVar(); emit(E.var ':=' '0')</td>
</tr>
</tbody>
</table>

- 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>
<tbody>
<tr>
<td>E → id op ids</td>
<td>E.var = freshVar(); emit('if' id.var relop id.var 'goto' nextStmt+1); emit(E.var ':=' '0'); emit('goto' nextStmt + 1); emit(E.var ':=' '1')</td>
</tr>
</tbody>
</table>

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false
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 if $x$ then $y$ else false;
- $(x \text{ or } y)$ is equivalent to if $x$ then true else $y$

Control Structures

- For every Boolean expression $B$, we attach two properties
  - falseLabel – target label for a jump when condition $B$ evaluates to false
  - trueLabel – target label for a jump when condition $B$ evaluates to true
- For every statement $S$ we attach a property
  - next – the label of the next code to execute after $S$
- Challenge
  - Compute falseLabel and trueLabel during code generation
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>

- 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 → ifB then S1</td>
<td>B.trueLabel = freshLabel(); B.falseLabel = S.next; S1.next = S.next; S.code = B.code</td>
</tr>
</tbody>
</table>

Boolean expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>B → not B1</td>
<td>B1.trueLabel = B.falseLabel; B1.falseLabel = B.trueLabel; B.code = B1.code</td>
</tr>
<tr>
<td>B → (B1)</td>
<td>B1.trueLabel = B.trueLabel; B1.falseLabel = B.falseLabel; B.code = B1.code</td>
</tr>
<tr>
<td>B → id1 relop id2</td>
<td>B.code = gen('if id1.var relop id2.var goto B.trueLabel')</td>
</tr>
<tr>
<td>B → true</td>
<td>B.code = gen('goto B.trueLabel')</td>
</tr>
<tr>
<td>B → false</td>
<td>B.code = gen('goto B.falseLabel')</td>
</tr>
</tbody>
</table>
### Boolean expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
</table>

- How can we determine the address of B1.falseLabel?
- Only possible after we know the code of B1 and all the code preceding B1

### Computing addresses for labels

- We used symbolic labels
- We need to compute their addresses
- We can compute addresses for the labels but it would require an additional 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

- Previous approach does not guarantee a single pass
  - The attribute grammar we had before is not S-attributed (e.g., next), and is not L-attributed.
- 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

Backpatching Boolean expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>B → B1 or B2</td>
<td>backpatch(B2, fakeList(M, instr)); B.trueList = merge(B1.trueList, B2.trueList); B.falseList = B2.falseList;</td>
</tr>
<tr>
<td>B → B1 and B2</td>
<td>backpatch(B2, trueList(M, instr)); B.trueList = B2.trueList; B.falseList = merge(B1.falseList, B2.falseList);</td>
</tr>
<tr>
<td>B → not B1</td>
<td>B.trueList = B1.trueList; B.falseList = B1.falseList;</td>
</tr>
<tr>
<td>B → (B1)</td>
<td>B.trueList = B1.trueList; B.falseList = B1.falseList;</td>
</tr>
<tr>
<td>B → id rel operators id</td>
<td>B.trueList = makeList(nextInstr); B.falseList = makeList(nextInstr + 1); emit('if id.var rel operators id.var goto _')</td>
</tr>
<tr>
<td>B → true</td>
<td>B.trueList = makeList(nextInstr); emit('goto _');</td>
</tr>
<tr>
<td>B → false</td>
<td>B.falseList = makeList(nextInstr); emit('goto _');</td>
</tr>
<tr>
<td>M → ε</td>
<td>M.instr = nextInstr;</td>
</tr>
</tbody>
</table>

Marker

- { M.instr = nextInstr; }
- Use M to obtain the address just before B2 code starts being generated
Example 37

\[ X < 150 \text{ or } x > 200 \text{ and } x \neq y \]

Example 38

\[ X < 150 \text{ or } x > 200 \text{ and } x \neq y \]

Example 39

\[ X < 150 \text{ or } x > 200 \text{ and } x \neq y \]

Example 40

\[ X < 150 \text{ or } x > 200 \text{ and } x \neq y \]
Example

\( X < 150 \text{ or } x > 200 \text{ and } x \neq y \)

Before backpatching

After backpatching by the production

Before backpatching

After backpatching by the production
Backpatching for statements

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{if} (B) M S_1$</td>
<td>$\text{backpatch}(B.\text{trueList}, M.\text{instr}); S.\text{nextList} = \text{merge}(B.\text{falseList}, S.\text{nextList});$</td>
</tr>
<tr>
<td>$S \rightarrow \text{if} (B) M S_1$</td>
<td>$\text{backpatch}(B.\text{falseList}, M.\text{instr}); S.\text{nextList} = \text{merge}(S.\text{nextList}, N.\text{nextList});$</td>
</tr>
<tr>
<td>$S \rightarrow \text{while} (B) M S_1$</td>
<td>$\text{backpatch}(S.\text{trueList}, M.\text{instr}); S.\text{nextList} = \text{merge}(B.\text{falseList}, S.\text{nextList});$</td>
</tr>
<tr>
<td>$S \rightarrow \text{set} A$</td>
<td>$S.\text{nextList} = M.\text{instr};$</td>
</tr>
<tr>
<td>$M \rightarrow s$</td>
<td>$M.\text{instr} = \text{nextInstr};$</td>
</tr>
<tr>
<td>$N \rightarrow v$</td>
<td>$N.\text{nextList} = \text{makeList}(\text{nextInstr});$</td>
</tr>
<tr>
<td>$L \rightarrow L. S$</td>
<td>$L.\text{nextList} = S.\text{nextList};$</td>
</tr>
<tr>
<td>$L \rightarrow S$</td>
<td>$L.\text{nextList} = S.\text{nextList};$</td>
</tr>
</tbody>
</table>

**Example**

if ($x < 150$ or $x > 200$ and $x = y$) $y = 200$

After backpatching by the production $B \rightarrow B_3$ or $M B_2$

After backpatching by the production $S \rightarrow \text{if} (B) M S_1$

**Procedures**

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

- 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”.

Coming up next...

- Activation Records

The End