CS406: Compilers

Spring 2020

Week 6: Semantic Actions and Code Generation

Case study - Semantic Analysis of Expressions

- Fully parenthesized expression (FPE)
  - Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
  - *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. (2 + (3 * 7))
    - So can ignore order-of-operations (PEMDAS rule)

Fully-parenthesized expression (FPE) – definition

- Recursive definition
  1. A number (integer in our example)
  2. *Open parenthesis* ‘(‘ followed by
     fully-parenthesized expression followed by
     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by
     fully-parenthesized expression followed by
     closed parenthesis ‘)’

Fully-parenthesized expression – notation

1. E -> INTLITERAL
2. E -> (E op E)
3. op -> ADD | SUB | MUL | DIV

A Hand-written Recursive Descent Parser for FPE

```
IsTerm(Scanner* s, TOKEN tok) { return s->GetNextToken() == tok;}

bool E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}

bool E2(Scanner* s) { return IsTerm(s, LPAREN) && E(s) && OP(s) && E(s) && IsTerm(s, RPAREN); }

bool OP(Scanner* s) {
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
               return true;
     return false;
}

bool E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     if(!E1(s)) {
               s->SetCurTokenSequence(prevToken);
               return E2(s);
     }
     return true;
}
```

Start the parser by invoking `E()`.
Value returned tells if the expression is FPE or not.

Building Abstract Syntax Trees

- Can build while parsing a fully parenthesized expression

  Via bottom-up building of the tree

- Create subtrees, make those subtrees left- and right-children of a newly created root.

  Modify recursive parser:

  1. If token == INTLITERAL, return a pointer to newly created node containing a number
  2. Else
     1. store pointers to nodes that are left- and right-expression subtrees
     2. Create a new node with value = ‘OP’

Building AST Bottom-up for FPE

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
      TreeNode* ret = NULL;
      TOKEN nxtToken = s->GetNextToken();
      if(nxtToken == tok)
                  ret = CreateTreeNode(nxtToken.val);
      return ret;
}

TreeNode* E1(Scanner* s) {
      return IsTerm(s, INTLITERAL);
}

TreeNode* E2(Scanner* s) {
      TOKEN nxtTok = s->GetNextToken();
      if(nxtTok == LPAREN) {
                  TreeNode* left = E(s);
                  if(!left) return NULL;
                  TreeNode* root  = OP(s);
                  if(!root) return NULL;
                  TreeNode* right = E(s)
                  if(!right) return NULL;
                  nxtTok = s->GetNextToken();
                  if(nxtTok != RPAREN); return NULL;
                              //set left and right as children of root.
                  return root;
      }
```

Building AST Bottom-up for FPE…

```
TreeNode* OP(Scanner* s) {
     TreeNode* ret = NULL;
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
                ret = CreateTreeNode(tok.val);
     return ret;
}

TreeNode* E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     TreeNode* ret = E1(s);
     if(!ret) {
               s->SetCurTokenSequence(prevToken);
               ret = E2(s);
     }
     return ret;
}
```

Start the parser by invoking `E()`.
Value returned is the root of the AST.

Identifying Semantic Actions for FPE Grammar

- What do we do when we see a `INTLITERAL`?
    - Create a `TreeNode`
    - Initialize it with a value (string equivalent of `INTLITERAL` in this case)
    - Return a pointer to `TreeNode`

Identifying Semantic Actions for FPE Grammar

- What do we do when we see an `E (parenthesized expression)`?
  - Create an AST node with two children. The node contains the binary operator `OP` stored as a string. Children point to roots of subtrees representing `E`.

Exercise

- *AST is a representation of:*
  a) Source program
  b) Collection of trees (one for arithmetic expr, declarations etc.)
  c) Tree data structure
  d) Syntax of the programming language

Exercise

1. *A symbol table contains names representing ______ of a program*

2. *Space required for a symbol table can be determined at compile time. True/False?*

Symbol Table

- A *symbol table* records
  - Symbolic names
  - Attributes of a name
    - E.g. type, scope, accessibility
- Used to manage declarations of symbols and their correct usage

Symbol Table – implementation strategy

- AST is the input to symbol table construction.
- Walk the tree and process declarations and usage
- Should accommodate:
  - Efficient retrieval of names
  - Frequent insertion and deletion of names

Symbol Table – implementation strategy

- AST is the input to symbol table construction
- Walk the tree, process declarations and usage
- Should accommodate:
  - Efficient retrieval of names
  - Frequent insertion and deletion of names

Symbol Table – example program

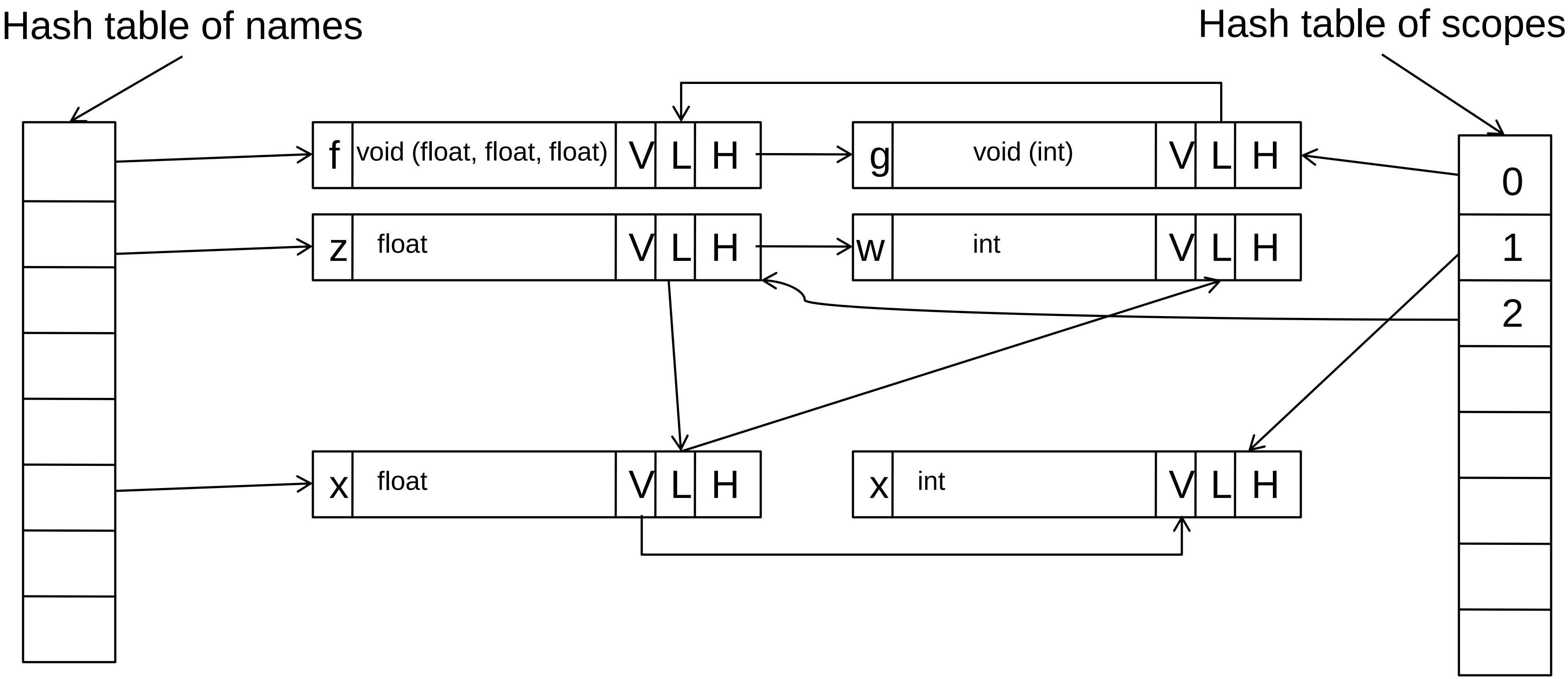
```
PROGRAM scope_test
BEGIN
//declarations
FUNCTION void f(float, float, float)
FUNCTION void g(int)

{
    INT w, x;
    {
       FLOAT x, z;
       f(x, w, z);
    }
    g(x);
}

END
```


* Example similar to that in Crafting a Compiler in C – Fischer, and LeBlanc.

Symbol Table – an implementation



* Example similar to that in Crafting a Compiler in C – Fischer, and LeBlanc.

Generating three-address code

- For project, will need to generate three-address code
  - op A, B, C //C = A op B
- Can do this directly or after building AST

Generating code from an AST

- Do a post-order walk of AST to generate code, pass generated code up

```
data_object generate_code() {
  //pre-processing code
  data_object lcode = left.generate_code();
  data_object rcode = right.generate_code();
  return generate_self(lcode, rcode);
}
```

- Important things to note:
  - A node generates code for its children before generating code for itself
  - Data object can contain code or other information

Generating code directly

- Generating code directly using semantic routines is very similar to generating code from the AST
  - Why?
  - Because post-order traversal is essentially what happens when you evaluate semantic actions as you pop them off stack
  - AST nodes are just semantic records
- To generate code directly, your semantic records should contain structures to hold the code as it's being built

Data objects

- Records various important info
  - The temporary storing the result of the current expression
  - Flags describing value in temporary
    - Constant, L-value, R-value
  - Code for expression

L-values vs. R-values

- L-values: addresses which can be stored to or loaded from
- R-values: data (often loaded from addresses)
  - Expressions operate on R-values
- Assignment statements:

  `L-value := R-value`

- Consider the statement `a := a`
  - the `a` on LHS refers to the memory location referred to by `a` and we store to that location
  - the `a` on RHS refers to data *stored in* memory location referred to by `a` so we will load from that location to produce the R-value

Temporaries

- Can be thought of as an unlimited pool of registers (with memory to be allocated at a later time)
- Need to declare them like variables
- Name should be something that cannot appear in the program (e.g., use illegal character as prefix)
- Memory must be allocated if address of temporary can be taken (e.g. $a := \&b$)
- Temporaries can hold either L-values or R-values

Simple cases

- Generating code for constants/literals
  - Store constant in temporary
  - Optional: pass up flag specifying this is a constant
- Generating code for identifiers
  - Generated code depends on whether identifier is used as L-value or R-value
    - Is this an address? Or data?
  - One solution: just pass identifier up to next level
    - Mark it as an L-value (it's not yet data!)
    - Generate code once we see how variable is used

Generating code for expressions

- Create a new temporary for result of expression
- Examine data-objects from subtrees
- If temporaries are L-values, load data from them into new temporaries
    - Generate code to perform operation
    - In project, no need to explicitly load (variables can be operands)
- If temporaries are constant, can perform operation immediately
    - No need to perform code generation!
- Store result in new temporary
    - Is this an L-value or an R-value?
- Return code for entire expression

Generating code for assignment

- Store value of temporary from RHS into address specified by temporary from LHS
  - Why does this work?
  - Because temporary for LHS holds an address
    - If LHS is an identifier, we just stored the address of it in temporary
    - If LHS is complex expression

```
int *p = &x
```

```
*(p + 1) = 7;
```

it *still* holds an address, even though the address was computed by an expression

Pointer operations

- So what do pointer operations do?
- Mess with L and R values
- & (address of operator): take L-value, and treat it as an R-value (without loading from it)

```
x = &a + 1;
```

- * (dereference operator): take R-value, and treat it as an L-value (an address)

```
*x = 7;
```

If statements

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

Generating code for ifs

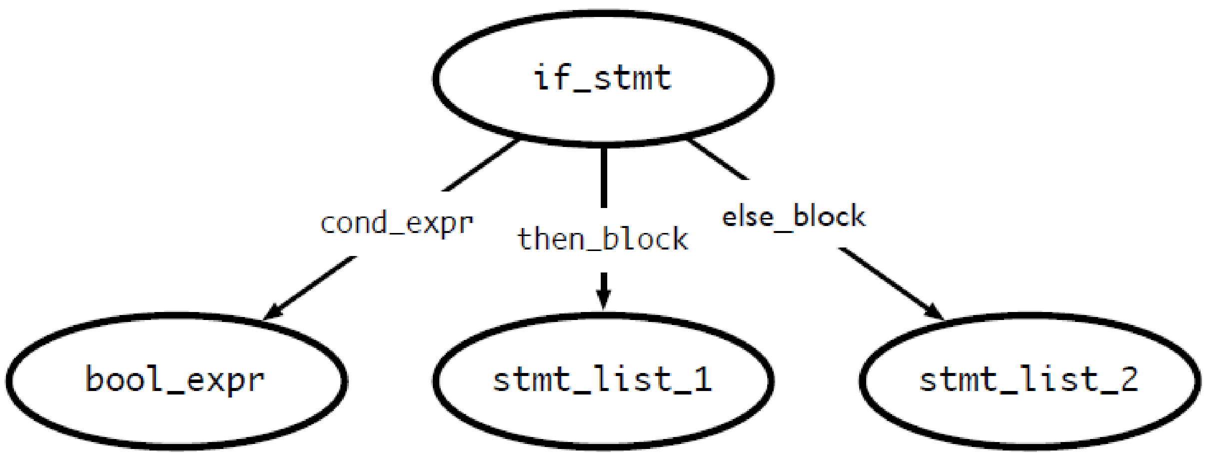
```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

```
  <code for bool_expr_1>
  j<!op> ELSE_1
  <code for stmt_list_1>
  jmp OUT_1
ELSE_1:
  <code for stmt_list_2>
OUT_1:
```

Notes on code generation

- The <op> in j<!op> is dependent on the type of comparison you are doing in <bool_expr>
- When you generate JUMP instructions, you should also generate the appropriate LABELs
- Remember: labels have to be unique!

If statements



Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Sections 2.7, 2.8
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 7, Chapter 8, Chapter 10