Semantic actions for declarations and expressions
Semantic actions

- **Semantic actions** are routines called as productions (or parts of productions) are recognized

- Actions work together to build up intermediate representations

- Conceptually think of this as follows:
  - Every non-terminal should have some information associated with it (code, declared variables, etc.)
  - Each child of a non-terminal can pass the information it has to its parent non-terminal, which uses the information from its children to build up more information
  - We call these **semantic records**
Semantic Records

- Data structures produced by semantic actions
- Associated with both non-terminals (code structures) and terminals (tokens/symbols)
- Standard organization: semantic stack
  - When you process a terminal (leaf in the parse tree), push its semantic record onto the stack
  - When you process a non-terminal:
    - Pop all the records associated with its children
    - Generate a record for the non-terminal
    - Push that record onto the stack
How do we manipulate stack?

- **Action-controlled**: actions directly manipulate stack (call push and pop)
- **Parser-controlled**: parser automatically manipulates stack
LR-parser controlled

- Shift operations push semantic records onto stack (describing the token)
- Reduce operations pop semantic records associated with symbols off stack, replace with semantic record associated with production
- Action routines do not see stack. Can refer to popped off records using handles
  - e.g., in yacc/bison, use $1, $2 etc. to refer to popped off records
Example of semantic actions

- Consider following grammar:

```
assign  →  ident := expr
expr    →  term addop term
term    →  ident | LIT
ident   →  ID
addop   →  + | –
```
Example of semantic actions

- In Bison (note that lexer returns values for each token through `yyylval`)

```plaintext
assign  → ident := expr {$$ = generateAssign($1, $3);}  
expr    → term addop term {$$ = generateExpr($1, $2, $3);}  
term    → ident {$$ = generateTerm($1);}  
        | LIT {$$ = generateTerm($1);}  
ident   → ID {$$ = $1;}  
addop   → + {$$ = ADD_OP;}  
        | − {$$ = SUB_OP;}  
```
Example of semantic stack

- Consider $a := b + 1$;
LL-controlled

- Even though LL parsers are not bottom up, semantic stack operates in basically the same way
- LL parsers take semantic actions as they encounter them while matching a production
- Add semantic action for a non-terminal at the end of the production
- Action for whole production gets processed after all of the intermediate parts of the production get processed
- In practice, this looks just like the LR-controlled stack
Example of semantic actions

- In ANTLR:
  assign returns [Code c]
  → ident := expr ${c = generateCode($ident.name, $expr.c);} 
  expr returns [Code c]
  → t1=term addop t2=term {
      $c = generateCode($t1.t, $t2.t, $addop.opType);
    }
  term returns [Term t]
  → ident ${t = generateTerm($ident.s);} 
    | LIT ${t = generateTerm($LIT.text);} 
  ident returns [String s] → ID ${s = $ID.text;}
  addop returns [OpType opType]
  → + ${opType = ADD_OP;} | – ${opType = SUB_OP;}

Monday, September 28, 15
Overview of declarations

- Symbol tables
- Action routines for simple declarations
- Action routines for advanced features
  - Constants
  - Enumerations
  - Arrays
  - Structs
  - Pointers
Symbol Tables

- Table of declarations, associated with each scope
  - Internal structure used by compiler – does not become code
- One entry for each variable declared
  - Store declaration attributes (e.g., name and type) – will discuss this in a few slides
- Table must be dynamic (why?)
- Possible implementations
  - Linear list (easy to implement, only good for small programs)
  - Binary search trees (better for large programs, but can still be slow)
  - Hash tables (best solution)
Handling declarations

- Declarations of variables, arrays, functions, etc.
- Create entry in symbol table
- Allocate space in *activation record*
  - Activation record stores information for a particular function call (arguments, return value, local variables, etc.)
  - Need to have space for all of this information
- Activation record stored on program stack
- We will discuss these in more detail when we get to functions
Simple declarations

• Declarations of simple types
  
  INT x;

  FLOAT f;

• Semantic action should
  
  • Get the type and name of identifier
  
  • Check to see if identifier is already in the symbol table
    
    • If it isn’t, add it, if it is, error
Simple declarations (cont.)

• How do we get the type and name of an identifier?
  \[
  \text{var}\_\text{decl} \rightarrow \text{var}\_\text{type} \ \text{id};
  \]
  \[
  \text{var}\_\text{type} \rightarrow \text{INT} \mid \text{FLOAT}
  \]
  \[
  \text{id} \rightarrow \text{IDENTIFIER}
  \]

• Where do we put the semantic actions?
Simple declarations (cont.)

- How do we get the type and name of an identifier?

  \[
  \text{var\_decl} \rightarrow \text{var\_type} \text{ id; } \{\text{currTable\_add($1, $2)} \}
  \]

  \[
  \text{var\_type} \rightarrow \text{INT} \{\$\$ = \text{INT}\} \mid \text{FLOAT} \{\$\$ = \text{FLOAT}\}
  \]

  \[
  \text{id} \rightarrow \text{IDENTIFIER} \{\$\$ = $1\}
  \]

- Where do we put the semantic actions?
  - Pass up the type
  - Pass up the variable name
  - Use both to create a symbol table entry
Managing symbol tables

- Maintain list of all symbol tables
- Maintain stack marking “current” symbol table
- Whenever you see a program block that allows declarations, create a new symbol table
  - Push onto stack as “current” symbol table
- When you see declaration, add to current symbol table
- When you exit a program block, pop current symbol table off stack
Managing symbol tables

• How do we manage multiple symbol tables?
  
  \[
  \text{func_decls} \rightarrow \text{func_decl} \text{ func_decls} \mid \text{empty}
  \]
  
  \[
  \text{func_decl} \rightarrow \text{any_type} \text{ id} \text{ BEGIN} \text{ decl}
  \text{ statement_list_list} \text{ END}
  \]

• Where do we put the semantic actions?
Managing symbol tables

- How do we manage multiple symbol tables?

```
func_decls → func_decl  func_decls | empty

func_decl → any_type id BEGIN
{symbolTableStack.push(currTable) currTable = new symbolTable();} decl statement_list
{currTable = symbolTableStack.pop()} END
```

- Where do we put the semantic actions?
Constants

• Constants
  • Symbol table needs a field to store constant value
  • In general, the constant value may not be known until runtime (static final int i = 2 + j;)
  • At compile time, we create code that allows the initialization expression to assign to the variable, then evaluate the expression at run-time
Arrays

• Fixed size (static) arrays
  
  int A[10];

• Store type and length of array

• When creating activation record, allocate enough space on stack for array

• What about variable size arrays?
  
  int A[M][N]

• Store information for a dope vector
  
  • Tracks dimensionality of array, size, location
  
  • Activation record stores dope vector
  
  • At runtime, allocate array at top of stack, fill in dope vector
Structs/classes

- Can have variables/methods declared inside, need extra symbol table
- Need to store visibility of members
- Complication: can create multiple instances of a struct or class!
- Need to store offset of each member in struct
Pointers

- Need to store type information and length of what it points to
  - Needed for pointer arithmetic
    
    ```
    int * a = &y;
    z = *(a + 1);
    ```

- Need to worry about forward declarations
  - The thing being pointed to may not have been declared yet
    
    ```
    Class Foo;
    Foo * head;
    Class Foo { ... };
    ```
Abstract syntax trees

- Tree representing structure of the program
- Built by semantic actions
- Some compilers skip this
- AST nodes
  - Represent program construct
  - Store important information about construct

```
    binary_op
      operator: +

    identifier
      "x"

    literal
      "10"
```
ASTs for References
Referencing identifiers

- Different behavior if identifier is used in a declaration vs. expression
  - If used in declaration, treat as before
  - If in expression, need to:
    - Check if it is symbol table
    - Create new AST node with pointer to symbol table entry
    - Note: may want to directly store type information in AST (or could look up in symbol table each time)
Referencing Literals

• What about if we see a literal?

  primary → INTLITERAL | FLOATLITERAL

• Create AST node for literal

• Store string representation of literal

  • “155”, “2.45” etc.

• At some point, this will be converted into actual representation of literal

  • For integers, may want to convert early (to do constant folding)

  • For floats, may want to wait (for compilation to different machines). Why?
More complex references

• Arrays
  • $A[i][j]$ is equivalent to
    $A + i*\text{dim}_1 + j$
  • Extract $\text{dim}_1$ from symbol table or dope vector

• Structs
  • $A.f$ is equivalent to
    &A + \text{offset}(f)$
  • Find $\text{offset}(f)$ in symbol table for declaration of record

• Strings
  • Complicated—depends on language
Expressions

• Three semantic actions needed
  • eval_binary (processes binary expressions)
    • Create AST node with two children, point to AST nodes created for left and right sides
  • eval_unary (processes unary expressions)
    • Create AST node with one child
  • process_op (determines type of operation)
    • Store operator in AST node
Expressions example

- $x + y + 5$
Expressions example

- $x + y + 5$

identifier "x"
Expressions example

- \( x + y + 5 \)
Expressions example

- \( x + y + 5 \)
Expressions example

- \( x + y + 5 \)
Expressions example

- $x + y + 5$
Generating three-address code

- For project, will need to generate three-address code
  - $\text{op } A, B, C \ // C = A \ \text{op} \ B$
- 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

```cpp
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
  • Code generation is context free
    • What does this mean?
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
- 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)
  \[
  *x = 7;
  \]