Semantic actions for 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)

• Build up semantic records by performing a bottom-up walk of the parse tree (as described in class)
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"
```
Referencing identifiers

- What do we return when we see an identifier?
  - 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?
  
  \[ \text{primary} \rightarrow \text{INTLITERAL} | \text{FLOATLITERAL} \]

- 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?
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

- $x + y + 5$
Expressions example

• $x + y + 5$

identifier "x"

identifier "y"
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
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:
  
  \[ \text{L-value} := \text{R-value} \]

- 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 \textit{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)
  \[
  *x = 7;
  \]