CS 536 Announcements for Tuesday, March 29, 2022

Midterm 2
- Wednesday, March 30, 7:30 – 9 pm
- B102 Van Vleck
- bring your student ID

Last Time
- building a predictive parser
- predictive parsing and syntax-directed translation

Today
- static semantic analysis
- name analysis
- exam review

Next Time
- Thursday, March 31 – no lecture
- Tuesday, April 4 – continue name analysis, start type checking

Static Semantic Analysis

Two phases
- name analysis (aka name resolution)
  - for each scope
    - process declarations
      - add entries to symbol table
      - report multiply-declared names
    - process statements
      - update IdNodes to point to appropriate symbol table entry
      - fix uses of undeclared variables
- type checking
  - process statements
    - use symbol table to find types of each expr & sub-expr
    - find type errors

![Diagram]

Printed copies of lecture overheads available at front
Why do we need this phase?

**Code generation**
- different operations use different instructions
  - consistent variable access
  - integer addition vs floating pt addition
  - operator overloading

**Optimization**
- symbol table entry serves to identify which variable is used
  - can help remove dead code (w/ further analysis)
  - note: pointers can make these tasks hard

**Error checking**

**Semantic error analysis**
For non-trivial programming languages, we run into fundamental undecidability problems:
- does the program halt?
- does the program crash?

Even with simplifying assumptions, sometimes infeasible in practice, as well
- combinations of thread interleavings
- inter-procedural data analysis

*In general - can’t guarantee the absence of errors*

**Goal of static semantic analysis**: catch some obvious errors
- undeclared identifiers
- multiply declared identifiers
- ill-typed terms
Name analysis

Associating IDs with their uses

Need to bind names before we can do type analysis

Questions to consider:
- What definitions do we need about identifiers?
- How do we bind definitions and uses together?

Symbol Table

= (structured) dictionary that binds a name to information we need

Each entry in the symbol table stores a set of attributes:
- kind - struct, variable, function, class
- type - int, int x string -> bool, struct
- nesting level
- runtime location (where in memory is it stored)

Symbol table operations
- insert entry
- lookup name
- add new subtable
- remove/forget a subtable

When do we do these operations?

Implementation considerations
- Efficiency of access is important
- Size unknown ahead of time -> need expansion to be graceful & efficient
- don't need to delete entries

⇒ use hash tables
Scoping

**scope** = block of code in which a name is visible/valid

**No scope (flat name scope)**

- assembly, FORTRAN name is visible throughout program

**Static/most-nested scope**

- block structure
- nested visibility
- easy to tell which def of a name applies
- new defs apply to local scope
- name scopes—limited region of definition

**Kinds of scoping**

**static** — can tell at compile time the correspondence between use & declaration

**dynamic** — correspondence determined at runtime

**Scoping issues to consider**

- Can the same name be used in multiple scopes?
- Can the same name be used multiple times in a single scope (if the names are of different kinds)?
- Where does declaration have to occur relative to use?
- How do we match up *uses to declarations*?
- What are the **boundaries of scopes**, e.g., are method/function parameters and local variables in the same scope?
Run the CYK algorithm to parse the input: D C C A B C
Example SDT on transformed grammar

CFG:
expr → term expr'
expr' → + term #1 expr' | ε
term → factor term'
term' → * factor #2 term' | ε
factor → #3 INTLIT | ( expr )

SDT actions:
#1 : termTrans = pop()
   exprTrans = pop()
   push(exprTrans + termTrans)
#2 : factorTrans = pop()
   termTrans = pop()
   push(termTrans * factorTrans)
#3 : push(INTLIT.val)

Parse table

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(  )</th>
<th>INTLIT</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr'</td>
<td>term expr'</td>
<td>term expr'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr' + term #1 expr'</td>
<td></td>
<td>ε</td>
<td>ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>term</td>
<td></td>
<td>factor term'</td>
<td>factor term'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td>ε</td>
<td>* factor #2 term'</td>
<td>ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td>( expr )</td>
<td>#3 INTLIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: 5 + 3*2 EOF

\[
\begin{align*}
\text{fT} &= 2 \\
\text{tT} &= 3 \\
\text{push } 3*2 &= 6 \\
\text{fT} &= 6 \\
\text{tT} &= 5 \\
\text{push } 5+6 &= 1
\end{align*}
\]