CS 536 Announcements for Monday, March 18, 2024

Midterm 2
- Thursday, March 21, 7:30 – 9 pm
- S429 Chemistry
- bring your student ID

Last Time
- static semantic analysis
- name analysis
  - symbol tables
  - scoping

Today
- name analysis
- exam review

Next Time
- type checking

Static Semantic Analysis

Two phases
- name analysis
- type checking

Name analysis
- for each scope
  - process declarations – add entries to symbol table
  - process statements – update IdNodes to point to appropriate symbol table entry
- each entry in symbol table keeps track of: kind, type, nesting level, runtime location
- identify errors
  - multiply-declared names
  - uses of undeclared variables
  - bad tuple accesses
  - bad declarations

Scoping
- **scope** = block of code in which a name is visible/valid
- kinds of scoping
  - **static** – correspondence between use & declaration made at compile time
  - **dynamic** – correspondence between use & declaration made at run time
Name analysis and tuples

Symbol tables and tuples

- Compiler needs to
  - for each field: determine type, size, and offset with the tuple
  - determine overall size of tuple
  - verify declarations and uses of something of a tuple type are valid

- Idea: each tuple type definition contains its own symbol table for its field declarations
  - associated with the main symbol table entry for that tuple's name

Relevant base grammar rules

```plaintext
decl ::= varDecl
      | fctnDecl
      | tupleDecl     // tuple defs only at top level
      ;
varDeclList ::= varDeclList varDecl
              | /* epsilon */
              ;
varDecl ::= type id DOT
          | TUPLE id id DOT
          ;
...  
tupleDecl ::= TUPLE id LCURLY tupleBody RCURLY DOT
            ;
tupleBody ::= tupleBody varDecl
         | varDecl
         ;
...  
type ::= INTEGER
       | LOGICAL
       | VOID
       ;
loc ::= id
      | loc COLON id
id ::= ID
    ;
```
Definition of a tuple type

tuple Point {
    integer x.
    integer y.
}.

tuple Color {
    integer r.
    integer g.
    integer b.
}.

tuple ColorPoint {
    tuple Color color.
    tuple Point point.
}.

Declaring a variable of type tuple

tuple Point pt.
tuple Color red.
tuple ColorPoint cpt.
Accessing fields of a tuple

\[
\begin{align*}
    pt:x &= 7. \\
    pt:y &= 8. \\
    pt:z &= 10. \\
    red:r &= 255. \\
    red:g &= 0. \\
    red:b &= 0. \\
    cpt:point:x &= pt.x. \\
    cpt:color:r &= red.r. \\
    cpt:color:g &= 34. 
\end{align*}
\]

If L child is an identifier
- check identifier
- get symbol table
- lookup

If L child is a colon-access
- recursively process L child
- if symbol table in
  - then
  - else

If R child is a tuple type
- then
- else
Name analysis: handling classes

Similar to handling aggregate data structures
  • also need to be able to search the class hierarchy

Idea:
Symbol table for each class with two nesting hierarchies
  1) for lexical scoping within methods
  2) for inheritance hierarchy

To resolve a name
  • first
  • then
CYK example

CFG
s → a C
| b a
a → A B
| C s
b → D
| ε

Convert to CNF

Run the CYK algorithm to parse the input: D C C A B C
FIRST/FOLLOW Example

Original CFG
expr → expr + term
  | term
term → term * factor
  | factor
factor → INTLIT
  | ( expr )

Transformed CFG
expr → term expr'
  expr' → + term expr' | ε
  term → factor term'
  term' → * factor term' | ε
  factor → INTLIT | ( expr )

<table>
<thead>
<tr>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td></td>
</tr>
<tr>
<td>expr'</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
</tr>
</tbody>
</table>

Parse table

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>expr'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Building the parse table

for each production x → α
  for each terminal T in FIRST(α)
    put α in table[x][T]
  if ε is in FIRST(α)
    for each terminal T in FOLLOW(x)
      put α in table[x][T]