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 symbol table entry for that tuple's name

Relevant base grammar rules

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

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

Recursively handle L child

If L child is an identifier
- check identifier has been declared of tuple type
- get symbol table for that tuple
- lookup R child in that sym tab

If L child is a colon-access
- recursively process L child
- if symbol table in access in null
  - then error
  - else look up R child in sym tab

If R child is a tuple type
- then set ref in : access to tuple's sym tab
- else set ref to null
Name analysis: handling classes

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

  to see if uses are of inherited fields & methods

Idea:

Symbol table for each class with two nesting hierarchies
  1) for lexical scoping within methods (i.e., "regular" sym tab)
  2) for inheritance hierarchy

  not just a list of hashable hierarchy not necessarily linear

To resolve a name
  • first look in lexical scoping sym tab (i.e., "regular" one)
  • then search inheritance hierarchy
CYK example

CFG

\[
\begin{align*}
  s & \rightarrow a\;C \\
  | & \quad b\;a \\
  a & \rightarrow A\;B \\
  | & \quad C\;s \\
  b & \rightarrow D \\
  | & \quad \varepsilon
\end{align*}
\]

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></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>INTLIT (</td>
<td>EOF )</td>
</tr>
<tr>
<td>expr'</td>
<td>+ ε</td>
<td>EOF (</td>
</tr>
<tr>
<td>term</td>
<td>INTLIT (</td>
<td>+ EOF )</td>
</tr>
<tr>
<td>term'</td>
<td>* ε</td>
<td>+ EOF )</td>
</tr>
<tr>
<td>factor</td>
<td>INTLIT (</td>
<td>* + EOF )</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>term exp'</td>
<td>term exp'</td>
</tr>
<tr>
<td>term</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>term'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>factor term'</td>
<td>factor term'</td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td></td>
<td></td>
<td>( expr )</td>
<td>INTLIT</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]