Topic 4: Abstract Syntax
Semantic Analysis

COS 320

Compiling Techniques

Princeton University
Spring 2015

Prof. David August
Abstract Syntax

Can write entire compiler in ML-YACC specification.

- Semantic actions would perform type checking and translation to assembly.
- Disadvantages:
  1. File becomes too large, difficult to manage.
  2. Program must be processed in order in which it is parsed. Impossible to do global/inter-procedural optimization.

Alternative: Separate parsing from remaining compiler phases.
• We have been looking at *concrete* parse trees.
  – Each internal node labeled with non-terminal.
  – Children labeled with symbols in RHS of production.

• Concrete parse trees inconvenient to use! Tree is cluttered with tokens containing no additional information.
  – Punctuation needed to specify structure when writing code, but
  – Tree structure itself cleanly describes program structure.
Parse Tree Example

\[
P \rightarrow (\ S \ ) \\
S \rightarrow S \ ; \ S \\
S \rightarrow \text{ID} := E
\]

\[
E \rightarrow \text{ID} \\
E \rightarrow \text{NUM} \\
E \rightarrow E + E \\
E \rightarrow E - E \\
E \rightarrow E \ast E \\
E \rightarrow E / E
\]

\[
(\ a := 4 ; b := 5 )
\]

```
       P
       /
      /
     S
    /\  /
   S  ;  S
   |
  S | |
  |
ID("a") := \ E
  |
  |
NUM(4)
```

```
|
ID("b") := \ E
  |
  |
NUM(4)
```

Type checker does not need "(" or ")" or ","
Parse Tree Example

Solution: generate *abstract parse tree* (abstract syntax tree) - similar to concrete parse tree, except redundant punctuation tokens left out.

```
CompoundStmt
   AssignStmt     AssignStmt
      ID("a")   NUM(4)     ID("b")   NUM(4)
```
Semantic Analysis: Symbol Tables

- Semantic Analysis Phase:
  - Type check AST to make sure each expression has correct type
  - Translate AST into IR trees

- Main data structure used by semantic analysis: *symbol table*
  - Contains entries mapping identifiers to their bindings (e.g. type)
  - As new type, variable, function declarations encountered, symbol table augmented with entries mapping identifiers to bindings.
  - When identifier subsequently used, symbol table consulted to find info about identifier.
  - When identifier goes out of scope, entries are removed.
Symbol Table Example

function f(b:int, c:int) =
  (print_int(b+c);
  let
    var j := b
    var a := "x"
  in
    print(a)
    print(j)
  end
  print_int(a)
)

\[ \sigma_0 = \{a \mapsto \text{int}\} \]

\[ \sigma_1 = \{b \mapsto \text{int}, c \mapsto \text{int}, a \mapsto \text{int}\} \]

\[ \sigma_2 = \{j \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}, a \mapsto \text{int}\} \]

\[ \sigma_3 = \{a \mapsto \text{string}, j \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}, a \mapsto \text{int}\} \]

\[ \sigma_1 = \{b \mapsto \text{int}, c \mapsto \text{int}, a \mapsto \text{int}\} \]

\[ \sigma_0 = \{a \mapsto \text{int}\} \]
Symbol Table Implementation

- Imperative Style: (side effects)
  - Global symbol table
  - When beginning-of-scope entered, entries added to table using side-effects. (old table destroyed)
  - When end-of-scope reached, auxiliary info used to remove previous additions. (old table reconstructed)

- Functional Style: (no side effects)
  - When beginning-of-scope entered, new environment created by adding to old one, but old table remains intact.
  - When end-of-scope reached, retrieve old table.
Symbol tables must permit fast lookup of identifiers.

- *Hash Tables* - an array of *buckets*
- *Bucket* - linked list of entries (each entry maps identifier to binding)

![Diagram](image)

- Suppose we wish to lookup entry for id \( i \) in symbol table:
  1. Apply *hash function* to key \( i \) to get array element \( j \in [0, n - 1] \).
  2. Traverse bucket in table[\( j \)] in order to find binding \( b \).
     (table[\( x \]): all entries whose keys hash to \( x \))
Functional Symbol Tables

Hash tables not efficient for functional symbol tables.

Insert \(a \rightarrow \text{string} \Rightarrow \) copy array, share buckets:

Old Symbol Table Array

\[
\begin{array}{c}
\text{i} \\
\text{a \rightarrow int}
\end{array}
\]

New Symbol Table Array

\[
\begin{array}{c}
\text{i} \\
\text{a \rightarrow string}
\end{array}
\]

Not feasible to copy array each time entry added to table.
Better method: use *binary search trees (BSTs)*.

- Functional additions easy.
- Need “less than” ordering to build tree.
  - Each node contains mapping from identifier (key) to binding.
  - Use string comparison for “less than” ordering.
  - For all nodes $n \in L$, $\text{key}(n) < \text{key}(l)$
  - For all nodes $n \in R$, $\text{key}(n) \geq \text{key}(l)$
Lookup:

- f -> int
- c -> int
- d -> int
- t -> int
- s -> int
Functional Symbol Table Example

Insert:

insert z \rightarrow \text{int}, create node z, copy all ancestors of z:

```
    c->int  f->int
     \downarrow       \downarrow
      t->int  d->int
          \   /   \
           /     \
          s->int t->int
               \   /   \
                \ /     \
                 z->int
```