CS415 Compilers

Context-Sensitive Analysis
Part 3

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Roadmap for the remainder of the course

- Project #2 – Bottom-up parser and compiler
  Due date Friday April 15
  Project intro video (29 minutes) available on canvas: My Media

- Sample solution for Homework #4 has been posted

- Posted "old" lecture videos on type systems, code generation, intermediate representations, and procedure abstractions to help with studying this material (Canvas, My Media). Not a replacement for attending lecture.

- Second midterm on Wednesday, April 6 (60 minutes in class)

- Final exam on May 10 , 1:00pm, (60 minutes in class)

Topics

- Regular expressions
- NFA and DFA
- Regular expressions to minimal DFA construction
- CFG
  - → Derivations
  - → Parse trees
  - → Ambiguity
- LL(1) parsing
  - → FIRST and FOLLOW sets
  - → Parse tables
  - → Recursive descent parsers
- LR(0) parsing
  - → LR(0) items
  - → LR(0) canonical collection and its construction
  - → ACTION and GOTO tables
  - → Shift/reduce and reduce/reduce conflicts

Types and Type Systems

Type: A set of values and meaningful operations on them

Types provide semantic "sanity checks" (consistency checks) and determine efficient implementations for data objects

Types help identify

- → errors, if an operator is applied to an incompatible operand
  - dereferencing of a non-pointer
  - adding a function to something
  - incorrect number of parameters to a procedure
  - ...
- → which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- → identification of polymorphic functions

Types and Type Systems

Type system: Each language construct (operator, expression, statement, …) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for

- basic types: `integer, char, real, boolean, typeError`
- constructed types, e.g., one-dimensional arrays: `array(lb, ub,` elem_type`)` , where elem_type is a type expression

A type checker implements a type system. It computes or "constructs" type expressions for each language construct.

Types and Type Systems

Example type inference rule:

$$\frac{E \vdash e_1 : \text{integer} \quad , \quad E \vdash e_2 : \text{integer}}{E \vdash e_1 + e_2 : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

3.0 + 1 or 1 + 3.0

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : ???}{E \vdash {*e} : ???}$$

where E is a type environment that maps constants and variables to their type expressions.

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : pointer(integer)}{E \vdash *e : integer}$$

where E is a type environment that maps constants and variables to their type expressions.

pointer(...) is now part of the type expression language such as array(...).

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : \text{pointer}(\beta)}{E \vdash {*e} : \beta}$$

where E is a type environment that maps constants and variables to their type expressions.

Type expressions may also contain type variables such as β. Type variables can denote any type expression.

Type variables are needed to express polymorphic types.

Example type inference rule address computation:

$$\frac{E \vdash e : ???}{E \vdash \&e : ???}$$

where E is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?

Types and Type Systems

Example type inference rule address computation:

$$\frac{E \vdash e : \text{integer}}{E \vdash \&e : \text{pointer(integer)}}$$

where E is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?

Types and Type Systems

Example type inference rule address computation:

$$\frac{E \vdash e : integer}{E \vdash \&e : pointer(integer)}$$

where E is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?

$$\frac{E \vdash e : \beta}{E \vdash \&e : pointer(\beta)}$$

Formal proof that a program can be typed correctly.

```
int a;                E = { a: integer }
. . .
. . . *(&a) + 3 . . .
```

Formal proof that a program can be typed correctly.

```
  int a;
 . . .
 . . . *(&a) + 3 . . .
```

E = { a: integer, 3: integer)

E |- a: integer

E |- (&a) : pointer (integer)

E |- *(&a): integer , E |- 3: integer

E |- *(&a) + 3 : integer

Type Names and Type Systems

Programmers may define their own types and give them names:

```
type my_int is int;
 …
int a;
my_int b;
 …
  … a + b …
```

Type names can also be part of the type expression language.

Note: type names and type variables are different!

Type Equivalence

Structural -- type equivalence: type names are expanded

Name -- type equivalence: type names are not expanded

Example:

```
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence:

name equivalence:

Type Equivalence

Structural -- type equivalence: type names are expanded

Name -- type equivalence: type names are not expanded

Example:

```
type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

Answer: structural equivalence: **(a, b, c, d, e)**

name equivalence: **(a); (b); (c, d, e);**

Syntax Directed Translation Scheme (SDT)

Revisit our type inference rule for "+".

```
exp   :  exp  '+' exp  {  if ($1 == TYPE_INT && $3 == TYPE_INT)
                              $$ = TYPE_INT;
                          else {
                              $$ = TYPE_INT;
                              printf("\n***Error: illegal operand types\n");
                          } }
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in attr.h . attr.c may contain helper functions.

The assignment of type expression C types to terminals and nonterminals of the grammar is done in parse.y.

Lexically-scoped Symbol Tables

§ 5.5 in EaC

The problem
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The interface
- insert(*name, level*) – creates record for *name* at *level*
- lookup(*name, level*) – returns pointer or index
- delete(*level*) – removes all names declared at *level*

Many implementation schemes have been proposed (see § B.4)
- We'll stay at the conceptual level
- Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names.
We'll see the corresponding run-time structures that are used to establish addressability later.

RUTGERS

Example

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure  r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
        }
        ... r ... s
    }
    ... q ...
}
```

```
B0: {
      int a, b, c
B1:   {
          int v, b, x, w
B2:           {
              int x, y, z
              ....
          }
B3:           {
              int x, a, v
              ...
          }
          ...
      }
    ...
  }
```

Example

procedure p {
int *a*, *b*, *c*
procedure *q* {
int *v*, *b*, *x*, *w*
procedure *r* {
int *x*, *y*, *z*
....
}
procedure *s* {
int *x*, *a*, *v*
...
}
... *r* ... *s*
}
... *q* ...
}

c

b, *w*

x, *a*, *v*

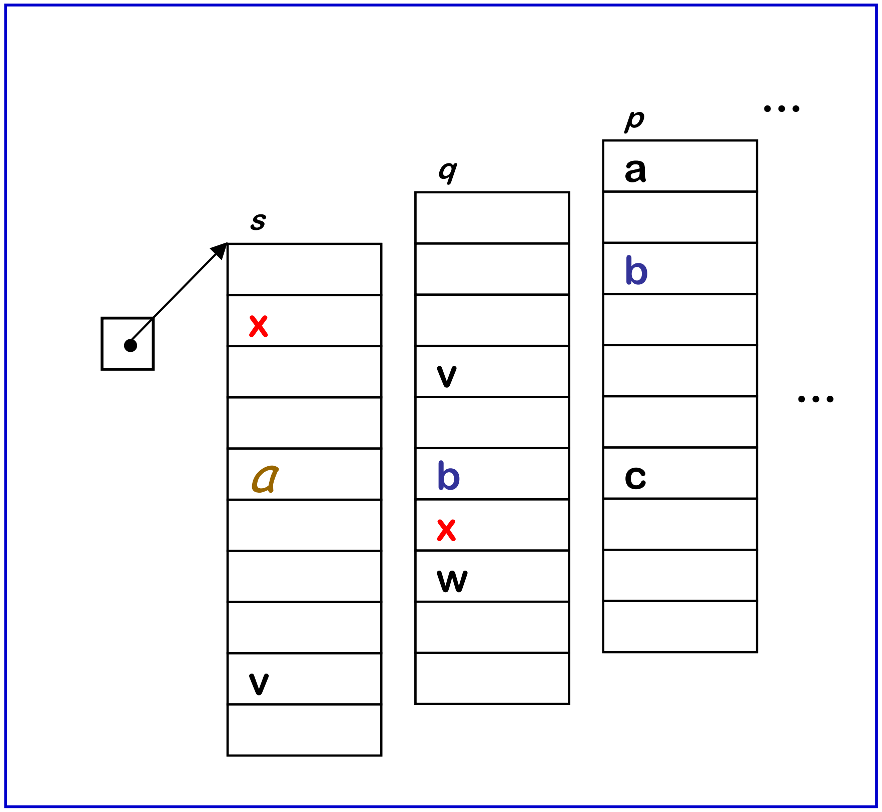
B0: {
int a_0, b_1, c_2
B1: {
int v_3, b_4, x_5, w_6
B2: {
int x_7, y_8, z_9
....
}
B3: {
int x_{10}, a_{11}, v_{12}
...
}
...
}
...
}

a_{11}, b_4, c_2,
v_{12}, w_6, x_{10},
no y or z

Picturing it as a series of
Algol-like procedures

Lexically-scoped Symbol Tables

High-level idea
- Create a new table for each scope
- Chain them together for lookup



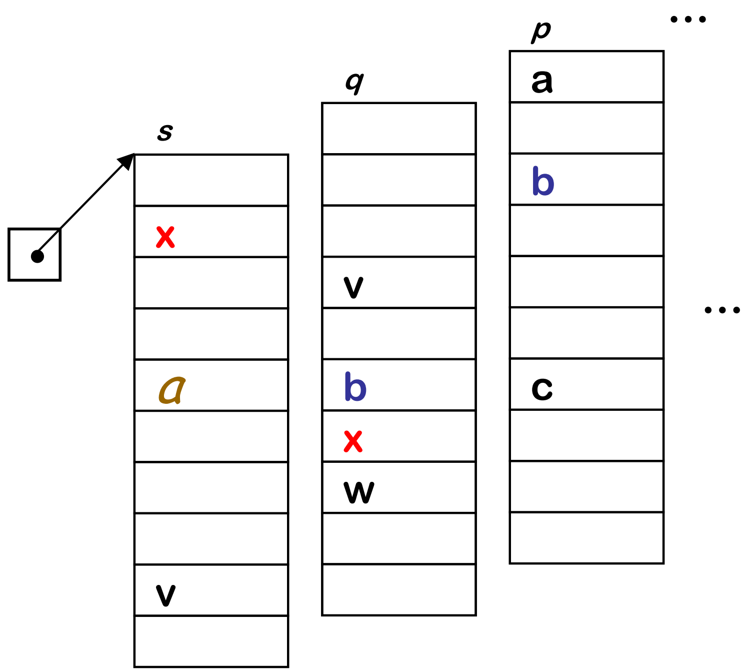
"Chain of tables" implementation
- *insert*() may need to create table
- it always inserts at current level
- *lookup*() walks chain of tables & returns first occurrence of name
- *delete*() throws away table for level *p*, if it is top table in the chain

Individual tables can be hash tables.

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



Remember

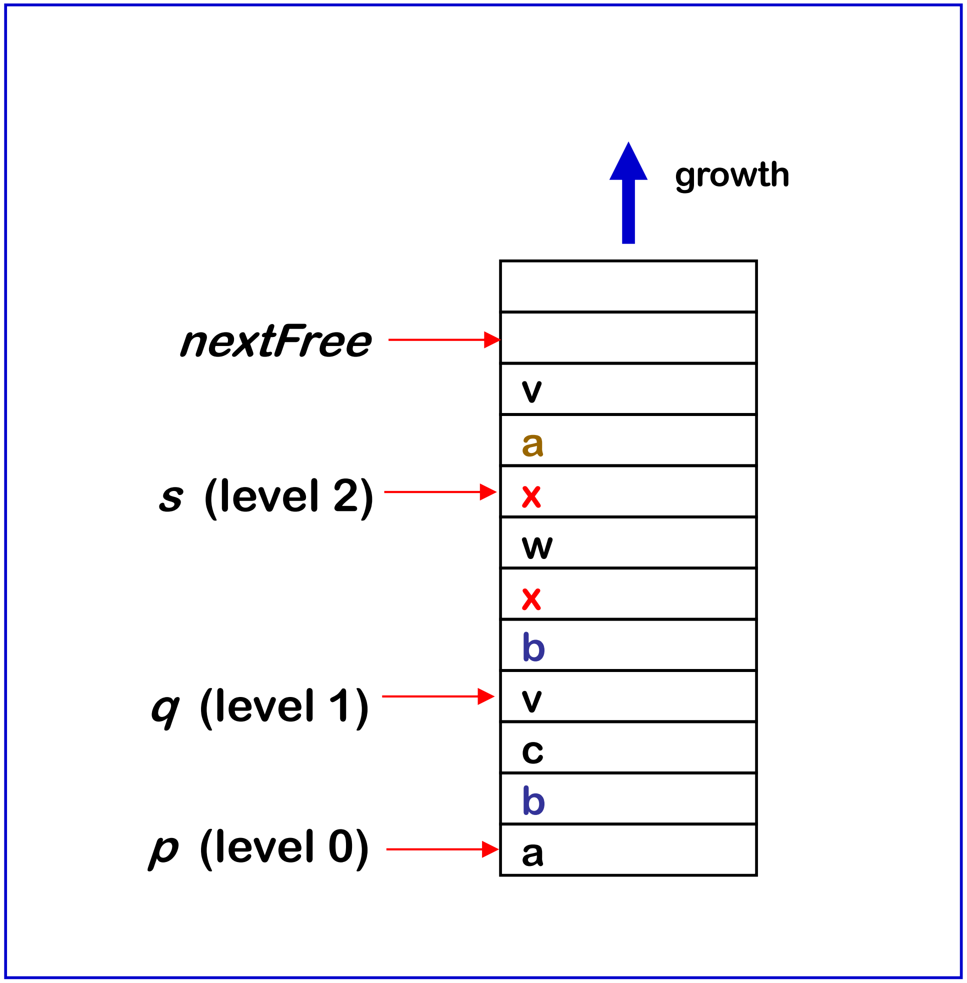
a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z

the names visible in s

If we add the subscripts, the relationship between the code and the table becomes clear

Implementing Lexically Scoped Symbol Tables

Stack organization



Implementation

- insert () creates new level pointer if needed and inserts at nextFree
- lookup () searches linearly from nextFree–1 forward
- delete () sets nextFree to the equal the start location of the level deleted.

Advantage

- Uses much less space

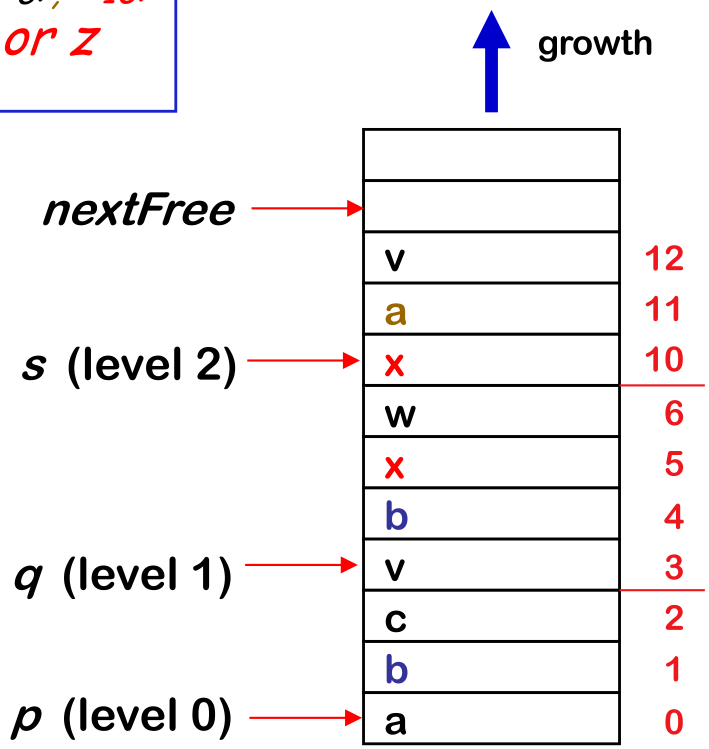
Disadvantage

- Lookups can be expensive

Implementing Lexically Scoped Symbol Tables

Stack organization

a_{11}, b_4, c_2,
v_{12}, w_6, x_{10},
no y or z



Implementation

- insert () creates new level pointer if needed and inserts at nextFree
- lookup () searches linearly from nextFree-1 down stack
- delete () sets nextFree to the equal the start location of the level deleted.

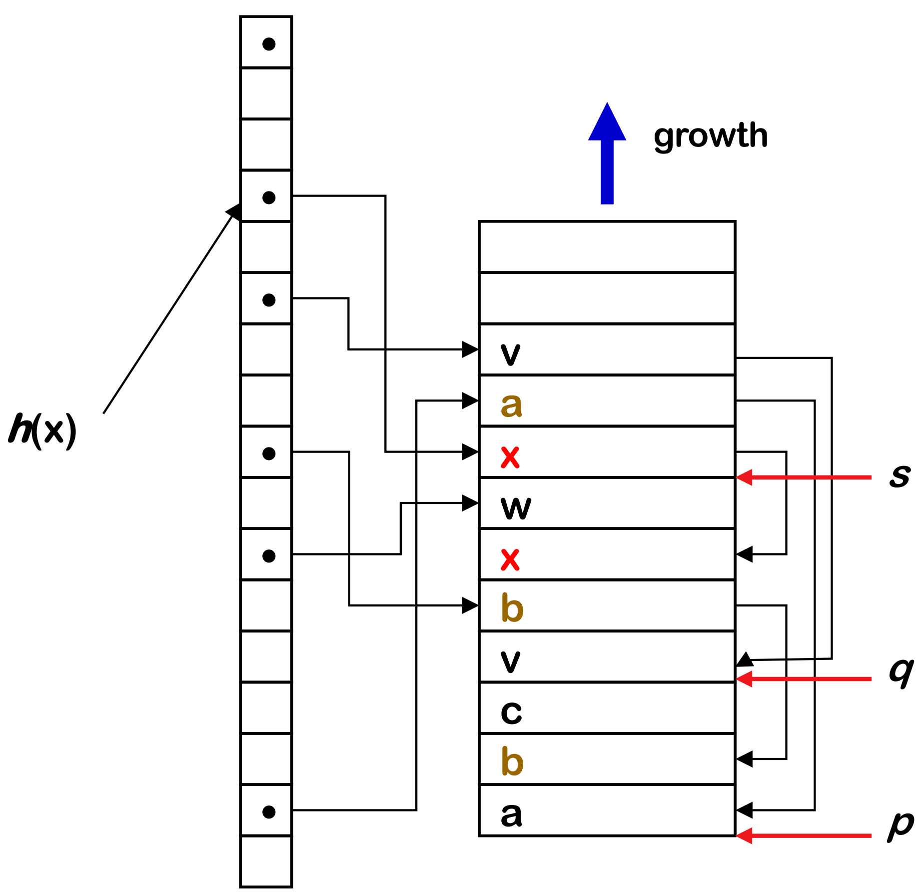
Advantage

- Uses <u>much</u> less space

Disadvantage

- Lookups can be expensive

Threaded stack organization



Implementation

- insert () puts new entry at the head of the list for the name
- lookup () goes direct to location
- delete () processes each element in level being deleted to remove from head of list

Advantage

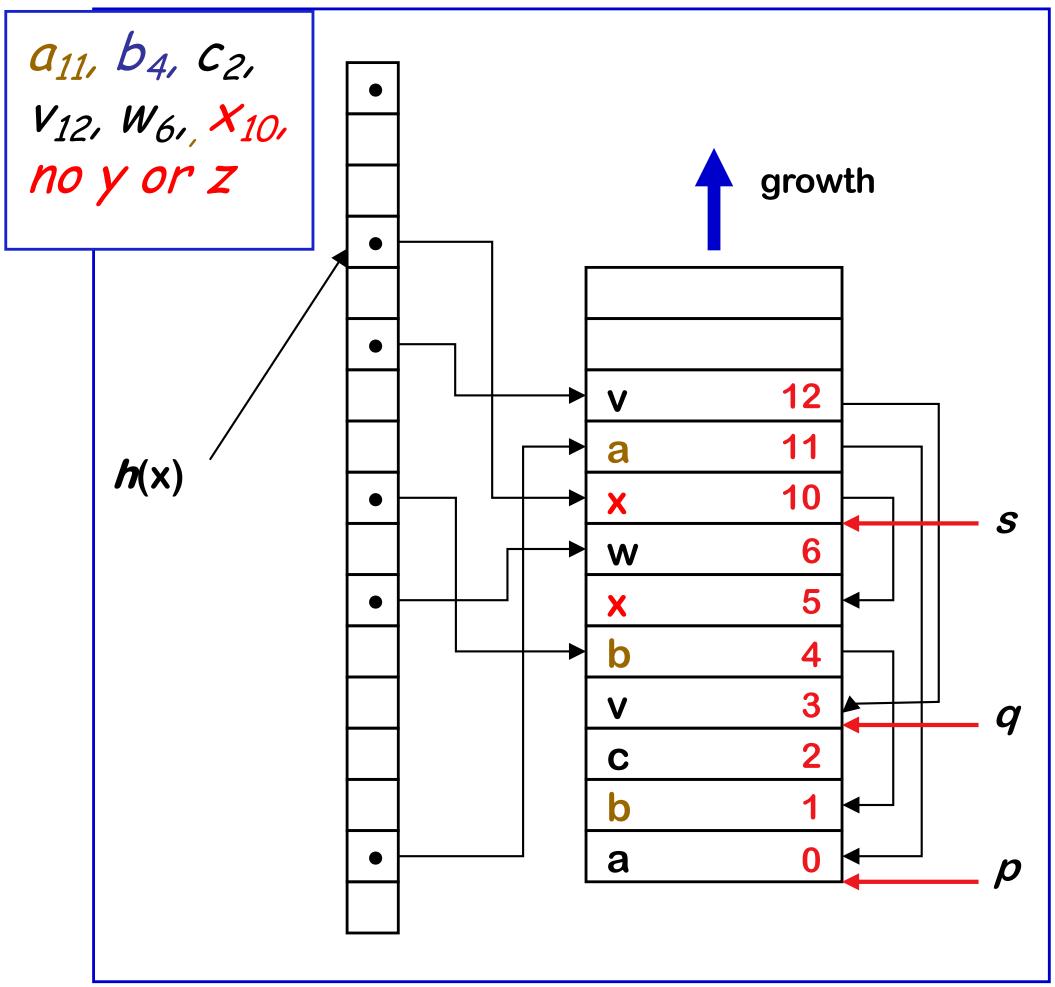
- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

Implementing Lexically Scoped Symbol Tables

Threaded stack organization



Implementation

- insert () puts new entry at the head of the list for the name
- lookup () goes direct to location
- delete () processes each element in level being deleted to remove from head of list

Advantage

- lookup is fast

Disadvantage

- delete takes time proportional to number of declared variables in level

Things to do and next class

Work on the project!

Code generation

Read EaC: Chapter 5

Intermediate representations

Read EaC: Chapter 5