CS415 Compilers

Context-Sensitive Analysis part 3

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

• Homework #6 has been posted.

• Project 2 has been posted.
Example on ilab: ~uli/cs415/examples/LexYacc

Relationship between practice and attribute grammars

Similarities
• Both rules & actions associated with productions
• Application order determined by tools
• (Somewhat) abstract names for symbols

Differences
• Actions applied as a unit; not true for AG rules
• Anything goes in *ad-hoc* actions; AG rules are (purely) functional
• AG rules are higher level than *ad-hoc* actions
A simple compiler

- Perform type checking (boolean and integer operations)
  → Error messages

- Generate ILOC code

- Lex & Yacc (C version)
You will need to define different types of attributes here to support code generation.

```c
#ifndef ATTR_H
#define ATTR_H

typedef union {int num; char *str;} tokentype;
typedef enum type_expression {TYPE_INT=0, TYPE_BOOL, TYPE_ERROR} Type_Expression;
typedef struct {
    Type_Expression type;
    int targetRegister;
} regInfo;
#endif
```
These definitions are needed for code generation.

#define NOLABEL -1
#define EMPTY 0
#define STATIC_AREA_ADDRESS 1024
#define MAX_VIRTUAL_REGISTERS 4096

typedef enum opcode_name {NOP=0, ADDI, ADD, SUBI, SUB, MULT, LOADI, LOAD, LOADAI, LOADAO, STORE, STOREAI, STOREAO, BR, CBR, CMPLT, CMPLE, CMPEQ, CMPNE, CMPGE, CMPGT, OUTPUTAI, AND_INSTR, OR_INSTR} Opcode_Name;

extern FILE *outfile;
extern int NextRegister();
extern int NextLabel();
extern int NextOffset(int units); /* units of 4 bytes */
extern void emitComment(char *comment);
extern void emit(int label_index, Opcode_Name opcode, int field1, int field2, int field3);
May need to add fields in SymTabEntry

typedef struct { /* need to augment this */
    char *name;
    int offset;
    Type_Expression type;
} SymTabEntry;

extern void InitSymbolTable();
extern SymTabEntry * lookup(char *name);
extern void insert(char *name, Type_Expression type, int offset);
extern void PrintSymbolTable();
Only shown partial code in parse.y

%union {tokentype token; regInfo targetReg;}
%token <token> ID ICONST
%type <targetReg> exp

writestmt: PRT '(' exp ')' { int printOffset = -4; /* default location for printing */

    sprintf(CommentBuffer, "Code for \"PRINT\" from offset %d", printOffset);
    emitComment(CommentBuffer);
    emit(NOLABEL, STOREAI, $3.targetRegister, 0, printOffset);
    emit(NOLABEL, OUTPUTAI, 0, printOffset, EMPTY);
}

exp : exp '+' exp { int newReg = NextRegister();

    if (! ($1.type == TYPE_INT) && ($3.type == TYPE_INT))) {
        printf("*** ERROR ***: Operator types must be integer.\n");
    }

    $$type = $1.type;
    $$targetRegister = newReg;
    emit(NOLABEL, ADD, $1.targetRegister, $3.targetRegister, newReg); }
exp: ...

| ID            | { /* BOGUS - needs to be fixed */
|               |   int newReg = NextRegister();
|               |   /* Symbol Table Lookup is missing */
|               |   int offset = NextOffset(4);
|               |   $$\cdot targetRegister = newReg;
|               |   $$\cdot type = TYPE_INT;
|               |   emit(NOLABEL, LOADAI, 0, offset, newReg); } |

| ICONST        | { int newReg = NextRegister();
|               |   $$\cdot targetRegister = newReg;
|               |   $$\cdot type = TYPE_INT;
|               |   emit(NOLABEL, LOADI, $1.num, newReg, EMPTY); } |
int main(int argc, char* argv[]) {
    printf("\n    CS415 Spring 2019 Compiler\n\n");
    outfile = fopen("iloc.out", "w");
    if (outfile == NULL) {
        printf("ERROR: cannot open output file \"iloc.out\".\n"); return -1; }
    CommentBuffer = (char *) malloc(5740);
   InitSymbolTable();
    printf("1\t");
    yyparse(); /* THIS IS THE CALL TO THE PARSER */
    printf("\n");
    PrintSymbolTable();
    fclose(outfile);
    return 1;
}
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
**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.
Example type inference rule:

\[
\begin{align*}
      E & \vdash e_1 : \text{integer} & E & \vdash e_2 : \text{integer} \\
      \hline \\
      E & \vdash (e_1 + e_2) : \text{integer}
\end{align*}
\]

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?

\[3.0 + 1 \quad \text{or} \quad 1 + 3.0\]
Example type inference rule pointer dereferencing:

\[
E \vdash e : ??? \\
\hline
E \vdash \ast e : ???
\]

where $E$ is a type environment that maps constants and variables to their type expressions.
Example type inference rule pointer dereferencing:

\[ E \vdash e : \text{pointer(integer)} \quad \text{where } E \text{ is a type environment that maps constants and variables to their type expressions.} \]

\[ E \vdash \ast e : \text{integer} \]

\textit{pointer(...)} is now part of the \textit{type expression language} such as \textit{array(...)}.
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 \(\beta\). Type variables can denote any type expression.

Type variables are needed to express polymorphic types.
Example type inference rule address computation:

\[
E \vdash e : \text{integer} \\
\underline{E \vdash \&e : ???}
\]

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

What about a polymorphic version of this rule?
Formal proof that a program can be typed correctly.

```c
int a;
...*(&a) + 3 ...
```
Programmers may define their own types and give them names:

```plaintext
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:

```plaintext
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:

```plaintext
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);
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
• \texttt{insert(name, level)} - creates record for \texttt{name} at \texttt{level}
• \texttt{lookup(name, level)} - returns pointer or index
• \texttt{delete(level)} - removes all names declared at \texttt{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.
Example

procedure p {
    int \( a, b, c \)

procedure q {
    int \( v, b, x, w \)

procedure r {
    int \( x, y, z \)

    \ldots \}
}

procedure s {
    int \( x, a, v \)

    \ldots \}

    \ldots \ r \ldots \ s

\ldots \ q \ldots 
}

B0: {
    int \( a, b, c \)

B1: {
    int \( v, b, x, w \)

B2: {
    int \( x, y, z \)

    \ldots \}
}

B3: {
    int \( x, a, v \)

    \ldots \}

    \ldots 

\ldots 
}
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
    }
}

B0: {  
    int a0, b1, c2
    B1: {  
        int v3, b4, x5, w6
        B2: {  
            int x7, y8, z9
            ...
        }
        B3: {  
            int x10, a11, v12
            ...
        }
        ...
    }
    ...
}
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

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

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

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 **much 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**
Threaded stack organization

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

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
Start working on the project!

Intermediate representations
Read EaC: Chapter 5.1 - 5.3

Code generation
Read EaC: Chapter 7.1 - 7.5