CS415 Compilers

Context-Sensitive Analysis
Part 2

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

• Project 1 and Quiz 2 have been graded
• Second project has been posted
Context-Sensitive Analysis

EaC Chapter 4
ALSU Chapter 5
Ad-hoc syntax-directed translation

- Associate pieces of code with each production
- At each reduction, the corresponding code is executed
- Allowing arbitrary code provides complete flexibility
  → Includes ability to do tasteless & bad things

To make this work

- Need names for attributes of each symbol on $lhs$ & $rhs$
  → Typically, one attribute passed through parser + arbitrary code (structures, globals, ...)
  → Yacc introduced $$, $1, $2, ... $n$, left to right
- Need an evaluation scheme
  → Fits nicely into $LR(1)$ parsing algorithm
→ You do not have to change the scanner (scan.l)

→ How to specify and use attributes in YACC?
  
  • Define attributes as types in attr.h

    typedef struct info_node {int a; int b} infonode;

  • Include type attribute name in %union in parse.y

    %union {tokentype token; infonode myinfo; ... }

  • Assign attributes in parse.y to

    - Terminals: %token <token> ID ICONST
    - Non-terminals: %type <myinfo> block variables procdecls cmpdstmt

  • Accessing attribute values in parse.y

    - use $$, $1, $2 ... etc. notation:

      block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt
             { $$a = $1.a + $2.a + $4.b;}
At each reduction, the corresponding code is executed.

→ Accessing attribute values in parse.y
  - use $$, $1, $2 ... etc. notation:

    \[
    \text{block : variables procdecls \{$2.b = \$1.b + 1;\} cmpdstmt} \\
    \{ \$$\text{.a = \$1.a + \$2.a + \$4.b;}\}
    \]

  - Implemented as

    \[
    \text{block : variables procdecls newsymbol cmpdstmt} \\
    \{ \$$\text{.a = \$1.a + \$2.a + \$4.b;}\}
    \]

    \[
    \text{newsymbol: \$ \{$2.b = \$1.b + 1;}\}
    \]
Summary: Is This Really “Ad-hoc”?

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();
%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); }

Only shown partial code in parse.y
exp: ...

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

| ICONST | { int newReg = NextRegister();
|       | $$\.targetRegister = newReg;
|       | $$\.type = TYPE\_INT;
|       | emit(NOLABEL, LOADI, $1.num, newReg, EMPTY); } |
```c
int main(int argc, char* argv[]) {
    printf("\n    CS415 Spring 2021 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(1832);
    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.
Things to do and next class

Continue working on the project!

More on type systems

Symbol table implementations

Intermediate representations
Read EaC: Chapter 5.1 - 5.3

Code generation
Read EaC: Chapter 7.1 - 7.5