Lec17 CS241

Graham Cooper

July 6th, 2015

Context-Sensitive Analysis (also called Semantic Analysis)

What properties of valid programs cannot be enforced by CFG's?

- Declaration before use
- Scope Correctness
- Type Checking
- etc...

To solve these, we need more complex programs
The next language class: Context-sensitive languages
(specified by context-sensitive grammars)

-Ad hoc approach actually more useful
- Traverse the parse tree

Eg. Parse tree
class Tree {
public:
- string rule;
- vector<Tree *> children;
};
The string rule is the grammar rule "term id" or the token type nad lexeme:
"id xyz"

Traversal Example: Print the tree

```
void print(Tree &t){
-- //print t.rule
-- foo(vector<Tree *> i ; iterator i = t.children.begin(); i != t.children.end; ++
-- -- print (**i);
-- }
}
```

Error Checking

What semantic analysis is needed for WLP4?
What errors need to be caught?

- variables and procedures, used but not declared
- variable/procedure duplicates
- scope?
- type errors
- For now, assume that wain is the only procedure

Decleration errors

multiple/missing declarations, how do we do this, we've seen this before
Construct a symbol table!!!!

- traverse the parse tree to collect all declarations.
- For each node corresponding to rule dcl $\rightarrow$ TYPE ID
- Extract the name and type and add to the symbol table
- if name is alrady in the symboltable, ERROR
- Multiple Declarations now checked

Traverse again!

- Check for rules of the form factor $\rightarrow$ ID and lvalue $\rightarrow$ ID
- if the ID's name is not in the symbol table, then ERROR
- undeclared variables are now checked

The above two passes could be merged.

Symbol Table Implementation - global variable

map <string, string> symbolTable
(name, type) But:

- doesn't take scope into account
- doesnt check procedure declerations

Issue:

```
int f() {
-- int x = 0;
-- return 1;
}

int wain (int a, int b){
-- int x = 0;
-- return 1;
}
```

This will give an error (only using the above two passes) and it should be OK!

Must forbid duplicate definitions in the same procedure, but allow them in different procedures

Also:

```
int f() { ... }
int f() { ... }
```

Must also forbid duplicate procedure definitions

Sol'n:

- Have a seperate symbol table for each procedure
- Have one "Top-Level" symbol table that collects all procedure names

map <string, map<string, string> > topSymtable;
(proc name, procs symboltable)
When traversing:
When the rule is
procedure → ...
main → ...
- these mean there is a new procedure
- make sure it is not already in the symbol table, and if not, create a new entry
- if not, create a new entry
- May want a global variable called "currentPRoc"
- Tells you which procedure you are currently in
- keep it up to date

For vars
- We store the declared type in the symbol table
Is there type into for procedures?
Yes - Signature - Note: All procedures in WLP4 return int, so the signature is the sequence of param types
- Store signature in top-level table

map< string, pair<vector <string>, map<string,string> > > topSymtbl;
(proc-name signature, local Symbol table)

To compute the signature:
Traverse nodes of the form - paramlist → dcl
- paramlist → dcl COMMA paramlist

Types

Why do programming languages have types?

- Prevent Errors
- allow us to assign an interpretation to the contents of some address and to remember the interpretation.

eg. int *a = NULL; a denotes a pair
a = 7; X attempt to stare an int where a pointer is denoted

To check type ocrrectness

- find a type for each var/expr
- ensure the "rules are paplied properly

Another tree traversal:

```
string typeOf(Tree &t){
-- for each c in t.children,
-- --compute typeOF(c)
-- determine typeOF(t) from types of children, based on t.rule
}
```

0.1 Example: ID

- get its type from symbol table

$$\frac{<\text{id.name}, \tau> \in \text{declarations}}{\text{id} : \tau}$$

```
string typeof(Tree &t){
-- if t.rule == "ID name"
-- -- return symtbl.lookup(name)
}
```

Singleton Productions

expr $\rightarrow$ term
term $\rightarrow$ factor
factor $\rightarrow$ id

$$\frac{\text{Type of LHS} = \text{type of RHS}}{\text{NUM: int ... NULL : int *}}$$

$$\frac{E : \tau}{(E) : \tau}$$

E has type τ then (E) has type τ