CS 241 — Lecture 17

Bartosz Antczak *Instructor: Gord Cormack* *March 7, 2017*

What we learned so far

A compiler is a program that takes a high-level language (which in this case is WLP4) and converts it to equivalent MIPS assembly language:

1. Compiler reads the WLP4 language using a **scanner**
2. Scanner produces tokens and passes them to the **parser**
3. Parser reads the program using L(1) and constructs a parse-tree

Now we are focusing on the next step, which is **context-sensitive analysis**.

17.1 Context-Sensitive Analysis

(Aside)

There is such thing as context-sensitive grammar, which is just like context-free grammar, but the rules are a little bit more general. They look like:

$$\alpha A \beta \to \alpha \gamma B \qquad \text{(expand } A \text{ to } \gamma \text{ only if preceded by } \alpha \text{ followed by } \beta\text{)}$$

(End of the aside)

Up to now, we made sure that what we were reading was syntactically correct. Now we're focused on the *semantics* of the language. If a program is syntactically valid, we now consider:

- **variables and procedures**: they can be undeclared, declared before use, have multiple declarations
- **types**: return value of procedures, parameter lists, operators
- **scope**: scope of variables in and out of procedures

Consider the following WLP4 program:

```
int wain(int a, int b) {
  int i = 10;
  int *p = NULL;

  i = i + i; // VALID
  i = i + p; // ERROR: assigning a pointer to an address
  i = p + i; // ERROR: same as above
  i = p + p; // ERROR: same as above
  i = p - p; // VALID
  p = i + i; // ERROR: assigning an integer to a pointer
  p = i + p; // VALID
  p = p + p; // ERROR: same as above
  p = p - p; // ERROR: same as above
```

```
  p = q; // ERROR: q is undefined
  return foo; // ERROR: foo is undefined
}
```

Recall that a pointer is structured as $\alpha + i$, where α represents a starting address and i is the offset. Observe that using a context-free grammar would not work. We must use context-sensitive analysis instead. We'll use it to solve these issues:

17.1.1 Variable Declaration Issues

We use a *symbol table*, similar to one that we used for MIPS. We'll track each variable's *name*, *location*, and *type*. We'll construct our symbol table and access a particular item's symbol table with `symboltable[s]`. Referring to the CFG rules of WLP4 for variables:

Rule	**Output for symbol table**
dcls $\rightarrow \varepsilon$	the symbol table is empty for dcls
$\text{dcls}_0 \rightarrow \text{dcls}_1$ dcl	merge the symbol tables of dcls_1 and dcl
$\text{dcls}_0 \rightarrow$ type ID	store the type and ID into the symbol table for dcls_0

After we construct our symbol table, we'll evaluate the type of all expressions using `typeof[expr]`

Rule	input for typeof
expr $\rightarrow$ term	typeof[expr] = typeof[term]
$\text{expr}_0 \rightarrow \text{expr}_1$ + term	`typeof[`expr_1`] == typeof[term] == int`; return int `typeof[`expr_1`] == int && typeof[term] == intptr)`; return intptr `typeof[expr1] == intptr && typeof[term] == int`; return intptr else, return invalid