# $L_2$ Language Concrete Syntax

## 1 $L_2$ Syntax

We're adding the following language features:

- Structs.
- Dynamic memory allocation.
- Pointers.

$$\begin{aligned} n \in \mathit{Integer} &::= \texttt{-}^?[0\text{–}9]^+ \\ id \in \mathit{Identifier} &::= [\text{a–zA–Z}]^+[0\text{–}9\text{a–zA–Z}]^\star \\ comment \in \mathit{Comment} &::= //.^\star\backslash\text{n} \\ typename \in \mathit{Typename} &::= \%[0\text{–}9\text{a–zA–Z}]^+ \end{aligned}$$

A *typename* is the name of a user-defined struct. Note that the set of valid typenames is disjoint from the set of valid identifiers, so they are easy to keep separate.

$$\begin{aligned} access \in \mathit{AccessPath} &::= id \mid access.id \\ aexp \in \mathit{ArithmeticExp} &::= n \mid access \mid \text{nil} \mid aexp + aexp \mid aexp - aexp \mid aexp * aexp \mid (aexp) \\ rexp \in \mathit{RelationalExp} &::= aexp < aexp \mid aexp = aexp \mid aexp <= aexp \mid rexp \;\&\&\; rexp \\ &\quad\mid rexp \;||\; rexp \mid\; !rexp \mid [\; rexp \;] \end{aligned}$$

We extend the notion of variable identifiers to *access paths*. An access path is an identifier optionally followed by a series of field dereferences. For example, foo.bar.baz is an access path starting from the identifier foo referencing a struct, dereferencing that struct's field bar to get a reference to another struct, and finally dereferencing that struct's field baz to get its value. We also add the expression nil, which is a reference to nothing (like a NULL pointer).

$$\begin{aligned} stmt \in \mathit{Statement} &::= assign \mid whileLoop \mid forLoop \mid cond \\ assign \in \mathit{Assignment} &::= access := aexp; \mid access := call; \mid access := \text{new } typename; \\ whileLoop \in \mathit{WhileLoop} &::= \text{while } rexp \;\{\; block \;\} \\ forLoop \in \mathit{ForLoop} &::= \text{for } id \text{ from } aexp \text{ to } aexp \;\{\; block \;\} \\ cond \in \mathit{Conditional} &::= \text{if } rexp \;\{\; block \;\} \mid \text{if } rexp \;\{\; block \;\} \text{ else } \{\; block \;\} \end{aligned}$$

Assignments can now modify an access path on the left-hand side. The right-hand side of an assignment can now be the dynamic memory allocation of a struct.

$$\begin{aligned} type \in \mathit{Type} &::= \text{int} \mid typename \\ decl \in \mathit{Declaration} &::= type\; id; \\ decls \in \mathit{Declarations} &::= \epsilon \mid decl\; decls \\ stmts \in \mathit{Statements} &::= \epsilon \mid stmt\; stmts \\ block \in \mathit{Block} &::= decls\; stmts \end{aligned}$$

User-defined structs are now valid types.

$$\begin{aligned} call \in \mathit{FunctionCall} &::= id(args) \mid id() \\ args \in \mathit{Arguments} &::= aexp \mid aexp, args \end{aligned}$$

$$\begin{aligned} \mathit{fundef} \in \mathit{FunctionDef} &::= \texttt{def}\ \mathit{id}(\mathit{optparams})\ :\ \mathit{type}\ \{\ \mathit{block}\ \texttt{return}\ \mathit{aexp};\ \} \\ \mathit{params} \in \mathit{Parameters} &::= \mathit{type}\ \mathit{id} \ \mid\ \mathit{type}\ \mathit{id},\ \mathit{params} \\ \mathit{optparams} \in \mathit{OptionalParameters} &::= \epsilon \ \mid\ \mathit{params} \\ \mathit{fundefs} \in \mathit{FunctionDefs} &::= \epsilon \ \mid\ \mathit{fundef}\ \mathit{fundefs} \end{aligned}$$

$$\begin{aligned} \mathit{typedef} \in \mathit{TypeDef} &::= \texttt{struct}\ \mathit{typename}\ \{\ \mathit{decls}\ \}; \\ \mathit{typedefs} \in \mathit{TypeDefs} &::= \epsilon \ \mid\ \mathit{typedef}\ \mathit{typedefs} \end{aligned}$$

A typedef is a struct declaration that provides the name of the struct along with the names and types of its fields.

$$\mathit{program} \in \mathit{Program} ::= \mathit{typedefs}\ \mathit{fundefs}\ \mathit{block}\ \texttt{output}\ \mathit{aexp};$$

A program consists of a (possibly empty) sequence of typedefs followed by a (possibly empty) sequence of function definitions followed by a block of statements followed by an output that will be printed by the program.

## 2 Example Program

```
struct %tree {
  int value;
  %tree left;
  %tree right;
};

def insert(%tree node, int value) : int {
  int dummy;
  if value <= node.value {
    if node.left = nil {
      node.left := new %tree;
      node.left.value := value;
    } else { dummy := insert(node.left, value); }
  } else {
    if node.right = nil {
      node.right := new %tree;
      node.right.value := value;
    } else { dummy := insert(node.right, value); }
  }
  return 0;
}

def find(%tree node, int value) : %tree {
  %tree retval;
  if node.value = value { retval := node; }
  else {
    if value < node.value {
      if node.left = nil { retval := nil; }
      else { retval := find(node.left, value); }
    }
```

```
    else {
      if node.right = nil { retval := nil; }
      else { retval := find(node.right, value); }
    }
  }
  return retval;
}

%tree root;
%tree node;
int dummy;

root := new %tree;
root.value := 0;

dummy := insert(root, -42);
dummy := insert(root, 42);

node := find(root, 42);

output node.value;
```