L_2 Language Concrete Syntax

1 L_2 Syntax

We're adding the following language features:

- Structs.
- Dynamic memory allocation.
- Pointers.

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

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.

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

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).

$$
\begin{aligned}
\mathit{stmt} \in \mathit{Statement} &::= \mathit{assign} \mid \mathit{loop} \mid \mathit{cond} \\
\mathit{assign} \in \mathit{Assignment} &::= \mathit{access} \texttt{ := } \mathit{aexp}\texttt{;} \mid \mathit{access} \texttt{ := } \mathit{call}\texttt{;} \mid \mathit{access} \texttt{ := new } \mathit{typename}\texttt{;} \\
\mathit{loop} \in \mathit{WhileLoop} &::= \texttt{while (}\mathit{rexp}\texttt{) \{ }\mathit{block}\texttt{ \}} \\
\mathit{cond} \in \mathit{Conditional} &::= \texttt{if (}\mathit{rexp}\texttt{) \{ }\mathit{block}\texttt{ \}} \mid \texttt{if (}\mathit{rexp}\texttt{) \{ }\mathit{block}\texttt{ \} else \{ }\mathit{block}\texttt{ \}}
\end{aligned}
$$

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.

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

User-defined structs are now valid types.

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

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

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

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

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

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 {
  if (value <= node.value) {
    if (node.left = nil) {
      node.left := new %tree;
      node.left.value := value;
    } else { insert(node.left, value); }
  } else {
    if (node.right = nil) {
      node.right := new %tree;
      node.right.value := value;
    } else { 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;
```