Language Overview : Group Members

Group Members

Submit, using handin, a file that lists the members of your group for this project. If you will be working alone, the file should contain only your name.

Language Overview

This document describes the language that we will be working with throughout the quarter: Extraordinarily Vexing Insipid Language. This language is similar in many respects to C, but limited in features.

Over the course of the term, you will implement an optimizing compiler for this language. Your first task is to familiarize yourself with the language.

Extraordinarily Vexing Insipid Language

The following grammar partially describes the language's syntax. In the EBNF below, non-terminals are typeset in **bold** font and terminals are typeset in `typewriter` font.

program	→	**types declarations functions**
types	→	{**type_declaration**}*
type_declaration	→	`struct id` { **nested_decl** } ;
nested_decl	→	**decl** ; {**decl** ;}*
decl	→	**type** `id`
type	→	`int` | `bool` | `struct id`
declarations	→	{**declaration**}*
declaration	→	**type id_list** ;
id_list	→	`id` {, `id`}*
functions	→	{**function**}*
function	→	`fun id` **parameters return_type** { **declarations statement_list** }
parameters	→	({**decl** {, **decl**}*}$_{opt}$)
return_type	→	**type** | `void`
statement	→	**block** | **assignment** | **print** | **read** | **conditional** | **loop** | **delete** | **ret** | **invocation**
block	→	{ **statement_list** }
statement_list	→	{**statement**}*
assignment	→	**lvalue** = **expression** ;
print	→	`print` **expression** {`endl`}$_{opt}$;
read	→	`read` **lvalue** ;
conditional	→	`if` (**expression**) **block** {`else` **block**}$_{opt}$
loop	→	`while` (**expression**) **block**
delete	→	`delete` **expression** ;
ret	→	`return` {**expression**}$_{opt}$;
invocation	→	`id` **arguments** ;
lvalue	→	`id` {. `id`}*
expression	→	**boolterm** {{&& | ||} **boolterm**}*
boolterm	→	**simple** {{== | < | > | ! = | <= | >=} **simple**}$_{opt}$
simple	→	**term** {{+ | −} **term**}*
term	→	**unary** {{∗ | /} **unary**}*
unary	→	{! | −}* **selector**

selector → **factor** {. `id`}*
factor → (**expression**) | `id` {**arguments**}$_{opt}$ | `number` | `true` | `false` | `new id` | `null`
arguments → ({**expression** {, **expression**}*}$_{opt}$)

The following rules complete the syntactic definition.

- A valid program is followed by an end-of-file indicator; extra text is not legal.
- The terminal (token) "id" represents a nonempty sequence (beginning with a letter) of letters and digits other than one of the keywords. Similarly, the terminal (token) "number" represents a nonempty sequence of digits.
- As is the case in most languages, tokens are formed by taking the longest possible sequences of constituent characters. For example, the input "abcd" represents a single identifier, not several identifiers. Whitespace (i.e., one or more blanks, tabs, or newlines) may precede or follow any token. E.g., "x=10" and "x = 10" are equivalent. Note that whitespace delimits tokens; e.g., "abc" is one token whereas "a bc" is two.
- A comment begins with "#" and consists of all characters up to a newline.
- Local declarations and parameters may hide global declarations (and functions), but local declarations cannot hide parameters.
- Structure names are in a separate namespace from variables and functions.

Semantics
The semantics for the language are given informally.

- Program execution begins in the function named `main` that takes no arguments and that returns an `int`. Every valid program must have such a function.
- The scope of each structure type is from the point of definition to the end of the file (this means that a structure type can only include elements of the primitive types and the structure types defined before it, though it should be able to include a member of its own type).
- The scope of each function is from the point of definition to the end of the file (though recursion must be supported, this restriction precludes mutual recursion).
- If and while have semantics equivalent to those of Java. They both require boolean guards.
- Assignment requires that the left-hand side and right-hand side have compatible types (equal in all cases except for `null`).
- A declaration with a structure type declares a reference to a structure (the structure itself must be dynamically allocated).
- `null` may be assigned to any variable of structure type.
- The `.` operator is used for field access (as in C and Java).
- All arguments are passed by value. For a structure reference, the reference itself is passed by value.
- `print` requires an integer argument and outputs it to standard out.
- `read` reads an integer value from standard in and stores it in the provided argument.
- `new` dynamically allocates a new structure, but does not initialize it, and evaluates to a reference to the newly allocated structure.
- `delete` deallocates the referenced structure.

- Arithmetic and relational operators require integer operands.
- Equality operators require operands of integer or structure type. The operands must have matching type. Structure references are compared by address (i.e., the references themselves are compared).
- Boolean operators require boolean operands.
- Boolean operators are *not* short-circuit.
- Each function with a non-void return type must return a valid value along all paths. Each function with a void return type must not return a value.