Computer Engineering 175
Phase IV: Type Checking

"The higher its type, the more rarely a thing succeeds."
Friedrich Nietzsche, *Thus Spake Zarathustra*

1 Overview

In this assignment, you will augment your parser to perform type checking for the Simple C language. This assignment is worth 20% of your project grade. Your program is due at 11:59 pm, Sunday, February 20th.

2 Type Checking

2.1 Overview

A very important issue in semantic checking is type checking. Type checking is the process of verifying that the operands to an operator are of the correct type. Each operator yields a value of a specified type based on the type of its operands. In Simple C, a type consists of a type specifier (`char`, `int`, or `struct`) along with optional declarators ("function returning *T*," "callback returning *T*," "array of *T*," and "pointer to *T*").

In Simple C, a value of type `char` may be ***promoted*** to type `int`, a value of type "array of *T*" may be promoted to type "pointer to *T*," and a value of type "function returning *T*" may be promoted to type "callback returning *T*." A type is a ***value*** type if it is not a structure type. Two types are ***compatible*** if (after any promotion) they are identical value types. An object is an ***lvalue*** if it refers to a location that can be used on the left-hand side of an assignment.

2.2 Semantic Rules

2.2.1 Statements

statement → { *declarations statements* }
| **return** *expression* ;
| **while** (*expression*) *statement*
| **for** (*assignment* ; *expression* ; *assignment*) *statement*
| **if** (*expression*) *statement*
| **if** (*expression*) *statement* **else** *statement*
| *assignment* ;

assignment → *expression* = *expression*
| *expression*

The type of the *expression* in a **return** statement must be compatible with the return type of the enclosing function [E1]. The type of an *expression* in a **while**, **if**, or **for** statement must be a value type [E2]. In an assignment statement the left-hand side must be an lvalue [E3], and the types of two sides must be compatible [E4].

2.2.2 Logical expressions

expression → *logical-and-expression*
| *expression* || *logical-and-expression*

logical-and-expression → *equality-expression*
| *logical-and-expression* && *equality-expression*

The type of each operand must be a value type, after any promotion [E4]. The result has type `int` and is not an lvalue. The types of the two operands need not be compatible.

2.2.3 Equality expressions

equality-expression → *relational-expression*
| *equality-expression* `==` *relational-expression*
| *equality-expression* `!=` *relational-expression*

The types of the left and right operands must be compatible, after any promotion [E4]. The result has type `int` and is not an lvalue.

2.2.4 Relational expressions

relational-expression → *additive-expression*
| *relational-expression* `<=` *additive-expression*
| *relational-expression* `>=` *additive-expression*
| *relational-expression* `<` *additive-expression*
| *relational-expression* `>` *additive-expression*

The types of the left and right operands must be compatible, after any promotion [E4]. The result has type `int` and is not an lvalue.

2.2.5 Additive expressions

additive-expression → *multiplicative-expression*
| *additive-expression* `+` *multiplicative-expression*
| *additive-expression* `-` *multiplicative-expression*

If both operands have type `int`, then the result has type `int`. If the left operand has type "pointer to *T*" and the right operand has type `int`, then the result has type "pointer to *T*." For addition only, if the left operand has type `int` and the right operand has type "pointer to *T*" then the result has type "pointer to *T*." For subtraction only, if both operands have type "pointer to *T*," where *T* is identical for both operands, then the result has type `int`. Otherwise, the result is an error [E4]. In all cases, operands undergo type promotion and the result is never an lvalue. Finally, if any operand has type "pointer to *T*," where *T* is a structure type, then *T* must be complete [E9].

2.2.6 Multiplicative expressions

multiplicative-expression → *prefix-expression*
| *multiplicative-expression* `*` *prefix-expression*
| *multiplicative-expression* `/` *prefix-expression*
| *multiplicative-expression* `%` *prefix-expression*

The types of both operands must be `int`, after any promotion [E4]. The result has type `int` and is not an lvalue.

2.2.7 Prefix expressions

prefix-expression → *postfix-expression*
| `-` *prefix-expression*
| `!` *prefix-expression*
| `&` *prefix-expression*
| `*` *prefix-expression*
| **sizeof** *prefix-expression*
| **sizeof** `(` *specifier pointers* `)`
| `(` *specifier pointers* `)` *prefix-expression*

The operand in a unary `*` expression must have type "pointer to *T*," after any promotion [E5] and must be complete if *T* is a structure type [E9]. The result has type *T* and is an lvalue. The operand in a unary `&` expression must be an lvalue [E3] and cannot have type "callback returning *T*" [E5]. If the operand has type *T*, then the result has type "pointer to *T*" and is not an lvalue. The operand does not undergo promotion.

The operand in a `!` expression must have a value type [E5], and the result has type `int`. The operand in a unary - expression must, after promotion, have type `int` [E5], and the result has type `int`. The operand of a **sizeof** expression, which does not undergo promotion, must not be a function type [E5] nor an incomplete structure type [E9]. The result of the expression has type `int`. In none of these cases is the result an lvalue.

For a type cast, the result type is that of *specifier*, along with any pointer declarators specified as part of *pointers*. The types of the result and the operand must (after any promotion) both be `int` or both be pointer types [E6]. The result is not an lvalue.

2.2.8 Postfix expressions

postfix-expression → *primary-expression*
| *postfix-expression* `[` *expression* `]`
| *postfix-expression* `(` *expression-list* `)`
| *postfix-expression* `(` `)`
| *postfix-expression* `.` **id**
| *postfix-expression* `->` **id**

expression-list → *expression*
| *expression* `,` *expression-list*

The left operand in an array reference expression must have type "pointer to *T*" and the *expression* must have type `int`, both after any promotion [E4], and *T* must be complete if *T* is a structure type [E9]. The result has type *T* and is an lvalue.

The function designator in a function call expression must have type "function returning *T*" or "callback returning *T*," and the result has type *T* [E7]. In addition, the arguments undergo promotion and must have value types. In addition, the number of parameters and arguments must agree and their types must be compatible, if the parameters have been specified [E8]. The result is not an lvalue.

The operand in a direct structure field reference must be a structure type, which must be complete, and the identifier must be a field of the structure [E4], in which case the type of the expression is the type of the identifier. The result is an lvalue if the expression is an lvalue and if the type of the identifier is not an array type.

The operand in an indirect structure field reference must be a pointer to a structure type (after any promotion) [E4], the structure type must be complete [E9], and the identifier must be a field of the structure [E4], in which case the type of the expression is the type of the identifier. The result is an lvalue if the type of the identifier is not an array type.

2.2.9 Primary expressions

primary-expression → **id**
| **num**
| **string**
| **character**
| `(` *expression* `)`

The type of an identifier is provided at the time of its declaration. An identifier is an lvalue if it refers to a scalar or callback (i.e., neither a function nor an array). A number has type `int` and is not an lvalue. A string constant has type "array of `char`" and is not an lvalue. A character constant has type `int` and is not an lvalue. The type of a parenthesized expression is that of the expression, and is an lvalue if the expression itself is an lvalue.

3 Assignment

You will write a semantic checker for Simple C by adding actions to your parser, using the given rules as a guide. Your compiler will be given only ***syntactically legal programs*** as input. Your compiler should indicate any errors by writing the appropriate error messages to the ***standard error*** (any output to standard output will be ignored):

E1. `invalid return type`

E2. `invalid type for test expression`

E3. `lvalue required in expression`

E4. `invalid operands to binary` *operator*

E5. `invalid operand to unary` *operator*

E6. `invalid operand in cast expression`

E7. `called object is not a function`

E8. `invalid arguments to called function`

E9. `using pointer to incomplete type`

4 Hints

Define and implement functions for abstractions such as type compatibility, checking if a type is `int`, a pointer, a value type, and test them thoroughly: these abstractions form the basis for most of the type checking rules. Implement the type checking rules in a bottom-up fashion. You will need to modify your parser so that the functions for the rules return the necessary type and lvalue information. A common implementation approach is to modify your functions for expressions so that they return the type and take a reference parameter through which to indicate whether the expression is an lvalue:

```
Type expression(bool &lvalue) {
    Type left = logicalAndExpression(lvalue);

    while (lookahead == OR) {
        match(OR);
        Type right = logicalAndExpression(lvalue);

        left = checkLogicalOr(left, right);
        lvalue = false;
    }

    return left;
}
```

A more advanced approach is to construct an abstract syntax tree during parsing as you perform the semantic checks. A tree node for an expression would contain its type and whether the expression is an lvalue. One could design an entire class hierarchy for the tree, such as statements (including if-statements, while-statements, etc.) and expressions (including addition, subtraction, etc.).