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 26th.

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`, `long`, or `void`) along with optional declarators ("function returning *T*," "array of *T*," and "pointer to *T*").

In Simple C, a value of type `char` may be ***promoted*** to type `int`, and a value of type "array of *T*" may be promoted to type "pointer to *T*." A type is ***numeric*** if it is either `int` or `long` after promotion, and is a ***predicate*** type if (after any promotion) it is numeric or "pointer to *T*." Two types are ***compatible*** if (after any promotion) they are both numeric, both are "pointer to *T*", where *T* is identical, or if one is "pointer to *T*" and the other is "pointer to `void`." 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 predicate 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 predicate 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 both be numeric or be identical predicate types, 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 the types of both operands are numeric, then the result has type `long` if either operand has type `long`, and has type `int` otherwise. If the left operand has type “pointer to *T*,” where *T* is not `void`, and the right operand has a numeric type, then the result has type “pointer to *T*.”

For addition only, if the left operand has a numeric type and the right operand has type “pointer to *T*,” where *T* is not `void`, then the result has type “pointer to *T*.” For subtraction only, if both operands have type “pointer to *T*,” where *T* is not `void` but is identical for both operands, then the result has type `long`. Otherwise, the result is an error [E4]. In all cases, operands undergo type promotion and the result is never an lvalue.

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 numeric [E4]. If either operand has type `long`, then the result has type `long`. Otherwise, the result has type `int`. The result is never an lvalue.

2.2.7 Prefix expressions

prefix-expression → *postfix-expression*
| `-` *prefix-expression*
| `!` *prefix-expression*
| `&` *prefix-expression*
| `*` *prefix-expression*
| **sizeof** *prefix-expression*

The operand in a unary `*` expression must have type “pointer to *T*,” after any promotion and where *T* is not `void` [E5]. The result has type *T* and is an lvalue. The operand in a unary `&` expression must be an lvalue [E3]. 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 predicate type [E5], and the result has type `int`. The operand in a unary `-` expression must, after promotion, have a numeric type [E5], and the result has the same promoted type. The operand of a **sizeof** expression must be a predicate type [E5]. The result of the expression has type `long`. In none of these cases is the result an lvalue.

2.2.8 Postfix expressions

postfix-expression	→	*primary-expression*
	|	*postfix-expression* `[` *expression* `]`

The left operand in an array reference expression must have type "pointer to *T*," where *T* is not `void`, and the *expression* must have a numeric type, both after any promotion [E4]. The result has type *T* and is an lvalue.

2.2.9 Primary expressions

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

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

The type of an identifier is provided at the time of its declaration. An identifier is an lvalue if it refers to a scalar (i.e., neither a function nor an array). If the value of a number can be represented as an `int`, then it has type `int`. Otherwise, it has type `long`. A number is not an lvalue. A string literal has type "array of `char`" and is not an lvalue. A character literal 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.

The identifier in a function call expression must have type "function returning *T*" and the result has type *T* [E6]. In addition, the arguments undergo promotion and must have predicate types. In addition, the number of parameters and arguments must agree and their types must be compatible, if the parameters have been specified [E7]. The result is not 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. `called object is not a function`

E7. `invalid arguments to called function`

4 Hints

Define and implement functions for abstractions such as type compatibility, checking if a type is numeric, a pointer, a predicate 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.).