Computer Engineering 175
Phase VI: Code Generation

> "This document describes the usage and input syntax of the Unix Vax-11 assembler As. As is designed for assembling code produced by the 'C' compiler; certain concessions have been made to handle code written directly by people, but in general little sympathy has been extended."
>
> *Berkeley Vax/Unix Assembler Reference Manual* (1983)

1 Overview

In this assignment, you will extend your compiler to generate code for a 32-bit Intel processor running the Linux operating system. This assignment is worth 20% of your project grade. Your program is due at 11:59 pm, Friday, March 15th.

2 Function Definitions and Statements

2.1 Overview

A function definition contains a sequence of statements and may yield a value through the **return** statement. Except as indicated, statements are executed in sequence. Statements are executed for their effect, and do not have values. Note that these rules are essentially the same as those of C, C++, and Java, and are provided for reference.

2.2 Semantic Rules

```
statement  →  { declarations statements }
           |  return expression ;
           |  while ( expression ) statement
           |  if ( expression ) statement
           |  if ( expression ) statement else statement
           |  expression = expression
           |  expression ;
```

The substatement of a **while** statement is executed repeatedly as long as the value of the *expression* is unequal to 0. Evaluation of the expression occurs before each execution of the substatement. In both forms of an **if** statement, the *expression* is evaluated and if its value is unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the value is equal to 0. In an assignment statement, the value of the right operand replaces that of the object referred to by the lvalue of the left operand. If necessary the value of the right operand is first converted.

A function returns to its caller by the **return** statement. The value of the *expression* is the return value of the function. The value is converted, as if by assignment, to the type returned by the function. Flowing off the end of a function is equivalent to a return with an undefined value.

3 Expressions

3.1 Overview

Each expression yields a value of a particular type. Unless the definition of an operator guarantees that its operands are evaluated in a particular order, an implementation is free to evaluate the operands in any order, or even to interleave their evaluation. Note that these rules are essentially the same as those of C, C++, and Java, and are provided for reference.

3.2 Semantic Rules

3.2.1 Logical expressions

$$\begin{array}{rcl}
\textit{logical-or-expression} & \rightarrow & \textit{logical-and-expression} \\
 & | & \textit{logical-or-expression}\ \texttt{||}\ \textit{logical-and-expression} \\
 & & \\
\textit{logical-and-expression} & \rightarrow & \textit{equality-expression} \\
 & | & \textit{logical-and-expression}\ \texttt{\&\&}\ \textit{equality-expression}
\end{array}$$

The || operator guarantees left-to-right evaluation: the first operand is evaluated; if it is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the value of the expression is 1, otherwise 0.

The && operator works similarly: the first operand is evaluated; it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the value of the expression is 0, otherwise 1.

3.2.2 Comparative expressions

$$\begin{array}{rcl}
\textit{equality-expression} & \rightarrow & \textit{relational-expression} \\
 & | & \textit{equality-expression}\ \texttt{==}\ \textit{relational-expression} \\
 & | & \textit{equality-expression}\ \texttt{!=}\ \textit{relational-expression} \\
 & & \\
\textit{relational-expression} & \rightarrow & \textit{additive-expression} \\
 & | & \textit{relational-expression}\ \texttt{<=}\ \textit{additive-expression} \\
 & | & \textit{relational-expression}\ \texttt{>=}\ \textit{additive-expression} \\
 & | & \textit{relational-expression}\ \texttt{<}\ \textit{additive-expression} \\
 & | & \textit{relational-expression}\ \texttt{>}\ \textit{additive-expression}
\end{array}$$

These operators all work similarly: the result of the expression is 0 if the comparison (in order listed above: equal to, not equal to, less than or equal to, greater than or equal to, less than, and greater than) is false, and 1 if it is true.

3.2.3 Additive expressions

$$\begin{array}{rcl}
\textit{additive-expression} & \rightarrow & \textit{multiplicative-expression} \\
 & | & \textit{additive-expression}\ \texttt{+}\ \textit{multiplicative-expression} \\
 & | & \textit{additive-expression}\ \texttt{-}\ \textit{multiplicative-expression}
\end{array}$$

The result of the + operator is the sum of its operands. A pointer to an object and an integer value may be added. The latter is converted to an address offset by multiplying it by the size of the object to which the pointer points. Similarly, the result of the - operator is the difference of its operands. An integer value may be subtracted from a pointer to an object, with the former converted to an address offset. A pointer to an object may be subtracted from another pointer. The result is the number of objects between the pointers.

3.2.4 Multiplicative expressions

$$\begin{array}{cl}
\rightarrow & \textit{prefix-expression} \\
| & \textit{multiplicative-expression} * \textit{prefix-expression} \\
| & \textit{multiplicative-expression}\ \texttt{/}\ \textit{prefix-expression} \\
| & \textit{multiplicative-expression}\ \texttt{\%}\ \textit{prefix-expression}
\end{array}$$

The result of the * operator is the product of its operands. Similarly, the result of the / operator is the quotient of its operands, and the result of the % operator is the remainder of its operands.

3.2.5 Prefix expressions

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

The result of the unary `-` operator is the negative of its operand. The result of the unary `!` operator is 1 if the value of its operand is equal to 0, and 0 otherwise. The unary $*$ operator denotes indirection and returns the object to which its operand points. The unary & operator yields the address of its operand. The **sizeof** operator yields the number of bytes required to store an object of the type of its operand. In a cast expression, the value of the expression is converted to the specified type as if by assignment.

3.2.6 Postfix expressions

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

The value of an array reference expression, E_1 `[` E_2 `]`, is identical (by definition) to $*$`((` E_1 `) + (` E_2 `))`.

3.2.7 Primary expressions

```
primary-expression  →  id ( expression-list )
                    |  id ( )
                    |  id
                    |  real
                    |  integer
                    |  string
                    |  ( expression )

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

The value of a parenthesized expression is simply that of the unadorned expression. The value of an identifier is simply the value of the object it identifies. The value of an integer is the value of the integer it represents, and the value of a real is the value of the real number it represents. The value of a string literal is the address of the first character in the string.

For function calls, arguments are passed by value. A function may therefore change the values of its parameters without affecting the values of the arguments. The order of evaluation of arguments is unspecified. An object of type "array of *T*" is converted to type "pointer to *T*." Recursive calls to any function are permitted. The result of a function call expression is the return value of the called function.

4 Assignment

You will write a code generator for Simple C using the given rules as a guide. Your compiler will only be given ***legal programs as input***. Your compiler should write valid 32-bit Intel assembly code to the ***standard output***. In the previous assignment, error messages were written to the standard error. Therefore, you do not need to change or remove the semantic checks already in place. Your generated assembly code can be assembled and linked using the native compiler as in the previous project.