The AppleCore Language Specification, v1.0

Robert L. Bocchino Jr.
Pittsburgh, PA

January 13, 2012

1 Introduction and Rationale

AppleCore is a C-like programming language for the Apple II series of computers. A cross-compiler from Java (e.g., running on a Mac or PC) to the Apple II is under development; eventually I would like to write an AppleCore compiler in AppleCore, thereby providing a compiler that works on the Apple II.

The idea of AppleCore is to provide a "core language for programming the Apple II." The motivation is as follows:

1. Programming the Apple II is both nostalgic and just plain fun.
2. The Apple II is a small machine. Therefore it needs a lightweight programming model (i.e., without complicated runtime overhead in space or time). The programming model must also provide (1) tight control over the layout of generated code and (2) easy integration with assembly-language support routines such as those provided by the System Monitor or Applesoft.
3. The native programming models on the Apple II are inadequate. Assembly language is powerful but painfully low-level. Applesoft, on the other hand, is a disaster. It lacks even the most basic abstraction required of a sane programming environment, including scoped loops, scoped functions with local variables, and variable names longer than *two letters*! Applesoft does have nice support for numeric and string input and output, so if you just want to write short programs that read and write things, it's handy. But for any but the tiniest programs, Applesoft code becomes an unreadable mess. In terms of control abstraction Applesoft is actually worse than assembly language, because it forces you to branch to numbered lines, whereas in assembly language you can `JMP` and `JSR` to symbolic labels with meaningful names.

The goal of AppleCore is to rectify this situation by providing a "low-level high-level" language (at about the same level of abstraction as C, but without C's features for structs) that's both useful and fun to program with. First, it's a compiled language, so it should run faster than Applesoft. Second, it's modular and supports separate compilation. The entire core language needs only a few kilobytes of support routines; additional routines can be linked in as needed. Third, AppleCore strikes a balance between power and efficiency. Notably, it supports integer arithmetic on fixed-size variables of up to 255 bytes. This power does come at a cost in terms of speed for single-byte arithmetic that could be done in registers; however, if that speed is needed, one can easily call into assembly language arithmetic routines. Fourth, it provides easy integration with assembly language routines: you can say things like "load 'A' into the accumulator and call $FDED" that are impossible to say directly in Applesoft. While you could use the `USR` function or a `CALL`, in either case you'd need to write some glue code in assembly language to get the value from the Applesoft program into the accumulator before the call.

The main things that AppleCore lacks, and that Applesoft provides, are automatic memory management and floating-point computation. I've left out memory management to keep things very simple. Basically, as a programmer

I can't live happily without control abstractions (loops and functions); but once I have them I'm perfectly content to synthesize data abstractions on my own. If this becomes too much of a pain, a library for memory management can be added without difficulty, after the fashion of C. (Applesoft actually provides garbage collection, but it works very poorly, causing the machine to seize up when the memory limit is reached.) I've left out floating-point computation mainly because I'm most interested in text processing and integer computations; but again, FP computations can be added without too much difficulty. In the mean time, one can fake FP computations by encoding FP literals as string data and calling into the Applesoft FP routines to convert from strings to FP numbers and back, and to compute using the FP numbers. Applesoft is actually quite good at this.

2 Lexical Structure

Before parsing a source file, the source text is separated into *tokens*. There are four classes of tokens: identifiers, keywords, constants, and symbols.

2.1 White Space

White space consists of any sequence of the following characters: space (ASCII SP, value $20), newline (ASCII NL, value $0A), carriage return (ASCII CR, value $0D), and horizontal tab (ASCII HT, value $09). Whitespace is syntactically relevant only in the following ways:

1. *Separation of tokens.* When a whitespace character or end of file appears immediately after a non-whitespace character, that signifies the end of a token.
2. *End of line.* An end-of-line sequence (`EOL`) is a comment terminator (see Section 2.6). As usual, the definition of `EOL` is platform-dependent: on the Apple II it is CR, on UNIX it is NL, and on Windows it is CR followed by NL.

The lexer may also count source lines (using `EOL`) to provide line numbers for error messages.

2.2 Identifiers

An identifier is a sequence of letters (including the underscore character) and digits that is not a keyword. The first character must be a letter. Upper- and lowercase letters are distinct.

2.3 Keywords

The following sequences of characters are reserved for use as keywords and may not be used as identifiers:

```
AND    CONST   DATA   DECR
ELSE   FN      IF     INCLUDE
INCR   NOT     OR     RETURN
SET    VAR     XOR    WHILE
```

Following Apple II tradition (mostly because the original Apple II had no support for lower-case letters), AppleCore keywords are uppercase. That means that the same words in lowercase (or a combination of upper- and lowercase) are not recognized as keywords, so those sequences of characters are available for use as identifiers.

2.4 Constants

There are three types of constants: integer constants, string constants, and character constants.

Integer constants. Integer constants may be written in decimal or hexadecimal form. A decimal integer constant consists of one or more decimal digits `0` through `9`. For example, `1` and `123` are valid integer constants in decimal form. A hexadecimal integer constant consists of a dollar sign `$` followed by by one or more hexadecimal digits `0` through `9` or `A` through `F`. For example, `$1`, `$A`, and `$00FF` are valid integer constants in hexadecimal form.

Integer constants represent values with up to 255 bytes of precision (i.e., in the range 0 through $2^{255 \cdot 8} - 1$, inclusive). It is a compile-time error to write an integer constant with a value larger than that.

String constants. A string constant is a sequence of characters enclosed in double quotes (`"..."`). A string constant represents a sequence of ASCII characters, one for each character appearing in the double-quotes, except that the character sequence `\$` has the special meaning described below.

Since not all ASCII characters have printable representations, an *escape sequence* may be used to represent an arbitrary ASCII value (printable or non-printable). An escape sequence consists of a backslash `\` followed by a dollar sign `$` and two hexadecimal digits. The whole sequence represents the single character with the ASCII value given by the digits. For example, the string constant

```
"Hello, world!\$0D"
```

represents a string consisting of the characters `Hello, world!` followed by a CR character. Similarly, the quote character `"` can be embedded in a string constant with the sequence `\$22`.

Character constants. A character constant consists of a single-quote character `'`, followed by a printable ASCII character, followed by another single-quote character. It represents the ASCII value associated with the character. For example, the constant `'A'` represents the value $41.

2.5 Symbols

AppleCore uses the symbols shown in Figure 1, each of which is a separate token.

2.6 Comments

The character # indicates a comment; all text to the next end of line (or end of file, if there is no end of line) are ignored by the lexer. Multi-line comments are indicated by preceding each line with #.

3 Syntax

The syntax description below uses the following conventions:

- The symbol * denotes zero or more instances of the entity preceding it.
- Italicized parentheses *( )* group the enclosed symbols and do not denote program text.
- Italicized brackets *[ ]* signify that the enclosed symbols are optional (i.e., they may occur zero or one time). They do not denote program text.
- The nonterminal *identifier* stands for any identifier as defined in Section 2.2.

Symbol	Meaning
`@`	Denotes the address of a variable
`^`	Denotes a 6502 register expression
`*`	Multiplication
`/`	Division
`+`	Addition
`-`	Negation (as unary operator); subtraction (as binary operator)
`<<`	Left shift
`>>`	Right shift
`>=`	Greater than or equal to
`<=`	Less than or equal to
`>`	Greater than
`<`	Less than
`=`	Equal to; assignment
`(` and `)`	Encloses function parameters and arguments, parenthesized expressions
`{` and `}`	Encloses statement blocks
`[` and `]`	Pointer dereference
`;`	Terminates declarations and statements
`:`	Separates variable declaration from size
`,`	Separates function parameters and arguments; separates index from size in dereference
`\`	Signifies unterminated string data

Figure 1: Symbols used in AppleCore syntax.

- The nonterminals *integer-const*, *string-const*, and *char-const* stand for integer, string, and character constants as defined in Section 2.4.
- The nonterminal *size* refers to an integer constant whose value is between 1 and 255 (inclusive).
- Text and symbols in `typewriter` font (including non-italicized parentheses and brackets) denote literal program text.

3.1 Source Files

The basic syntactic unit of an AppleCore program is a *source file*, i.e., an input file presented to the AppleCore compiler for compilation. The compiler translates the source file into an assembly file which is then linked with other assembly files as discussed in Section 5.1 to form a complete executable program.

An AppleCore source file is given by zero or more declarations (Section 3.2):

$$\textit{source-file} ::= \textit{decl}^*$$

3.2 Declarations

A declaration is a constant declaration, a data declaration, a variable declaration, a function declaration, or an include declaration:

$$\textit{decl} ::= \textit{const-decl} \mid \textit{data-decl} \mid \textit{var-decl} \mid \textit{fn-decl} \mid \textit{include-decl}$$

Constant declarations. A constant declaration consists of the keyword `CONST`, an identifier, an expression, and a terminating semicolon:

$$\textit{const-decl} ::= \texttt{CONST}\ \textit{identifier}\ \textit{expr}\ \texttt{;}$$

Data declarations. A data declaration consists of the keyword `DATA`, an optional identifier representing a label for the data, an expression or a string constant, and a terminating semicolon. A backslash may optionally follow the string constant, indicating that the string is unterminated (see Section 5.2).

data-decl ::= `DATA` *[identifier] (expr* | *(string-const [* `\` *]))* `;`

Variable declarations. A variable declaration consists of the keyword `VAR`, an identifier representing the variable name, a signed or unsigned size, an optional initializer expression, and a terminating semicolon:

var-decl ::= `VAR` *identifier* `:` *size [* `S` *] [* `=` *expr]* `;`

Function declarations. A function declaration consists of the keyword `FN`, an optional signed or unsigned size, an identifier representing the function name, the function parameters enclosed in parentheses, and the function body:

fn-decl ::= `FN` *[* `:` *size [* `S` *]] identifier* `(` *fn-params* `)` *fn-body*

The function parameters are a comma-separated list of zero or more parameters:

fn-params ::= *[fn-param (* `,` *fn-param)** *]*

A parameter consists of an identifier representing the parameter name and a size:

fn-param ::= *identifier* `:` *size [* `S` *]*

A function body is either zero or more variable declarations and statements enclosed in braces, or a semicolon indicating an externally defined function:

fn-body ::= { *var-decl** *stmt** } | `;`

Include declarations. An include declaration consists of the keyword `INCLUDE`, a string constant, and a terminating semicolon:

include-decl ::= `INCLUDE` *string-const* `;`

3.3 Statements

A statement is an if statement, a while statement, a set statement, a call statement, and increment statement, a decrement statement, return statement, or a block statement:

stmt ::= *if-stmt* | *while-stmt* | *set-stmt* | *call-stmt* | *incr-stmt* | *decr-stmt* | *return-stmt* | *block-stmt*

If statements. An if statement consists of the keyword `IF`, a conditional expression enclosed in parentheses, a statement to execute if the condition is true, and optionally the keyword `ELSE` followed by a statement to execute if the condition is false:

if-stmt ::= `IF` `(` *expr* `)` *stmt [* `ELSE` *stmt]*

While statements. A while statement consists of the keyword `WHILE`, a test expression enclosed in parentheses, and a statement to execute as long as the condition is true:

while-stmt ::= `WHILE` `(` *expr* `)` *stmt*

Set statements. A set statement consists of the keyword `SET`, an lvalue expression, an equals sign, a right-hand-side expression, and a terminating semicolon:

set-stmt ::= `SET` *lvalue-expr* `=` *expr* `;`

Call statements. A call statement consists of a call expression followed by a terminating semicolon:

call-stmt ::= *call-expr* `;`

Increment statements. An increment statement consists of the keyword `INCR`, an expression, and a terminating semicolon:

incr-stmt ::= `INCR` *expr* `;`

Decrement statements. n decrement statement consists of the keyword `DECR`, an expression, and a terminating semicolon:

decr-stmt ::= `DECR` *expr* `;`

Return statements. A return statement consists of the keyword `RETURN`, an optional expression, and a terminating semicolon:

return-stmt ::= `RETURN` *[expr]* `;`

Block statements. A block statement consists of zero or more statements enclosed in braces:

block-stmt ::= { *stmt** }

3.4 Expressions

An expression is an lvalue expression, a numeric constant, a call expression, a binary operation expression, a unary operation expression, or a parentheses expression.

expr ::= *lvalue-expr* | *numeric-const* | *call-expr* | *binop-expr* | *unop-expr* | *parens-expr*

An lvalue expression is an expression that may appear on the left-hand side of a set statement. It is an identifier, an indexed expression, or a register expression.

lvalue-expr ::= *identifier* | *indexed-expr* | *register-expr*

Numeric constants. A numeric constant is an integer or character constant:

numeric-const ::= *integer-const* | *char-const*

Call expressions. A call expression consists of an expression followed by an argument list enclosed in parentheses:

call-expr ::= *expr* `(` *[expr (, expr)** *]* `)`

Indexed expressions. An indexed expression consists of a base expression, an offset expression, and a size:

indexed-expr ::= *expr* `[` *expr , size* `]`

Register expressions. A register expression consists of a caret character followed by a 6502 register name:

register-expr ::= `^` *(* `A` | `X` | `Y` | `P` | `S` *)*

Binary operation expressions. A binary operation expression consists of a left-hand-side expression, a binary operator, and a right-hand-side expression:

binop-expr ::= *expr binop expr*

binop ::= > | < | <= | >= | `AND` | `OR` | `XOR` | + | − | * | / | << | >> | =

Parsing of binary operation expressions is disambiguated using the following precedence rules:

1. In any sequence *expr-1 binop-1 expr-2 binop-2 expr-3*, the implied parentheses go around the first operation unless the second operation has a strictly higher precedence. There are five levels of precedence, from highest to lowest: (1) << and >>; (2) $\star$ and \; (3) + and −; (4) =, >, <, <=, and >=; and (5) `AND`, `OR`, and `XOR`.

2. Binary operators bind more weakly than any other expression combinators. For example, `A+B()` is parsed as `A+(B())` and not `(A+B)()`; and `-5+3` is parsed as `(-5)+3` and not `-(5+3)`.

Unary operation expressions. A unary operation expression consists of a unary operator followed by an expression:

unop-expr ::= *unop expr*

unop ::= `@` | `NOT` | `-`

Parentheses expressions. A parentheses expression is an expression surrounded by parentheses:

parens-expr ::= `(` *expr* `)`

4 Semantic Checking

The compiler checks every source file for conformance to the semantic rules stated below. A semantically invalid program (i.e., one that violates one of these rules) generates a compile-time error and causes compilation to halt. Only semantically valid files go on to code generation as described in Section 5.

4.1 Representation of Values

The AppleCore language supports two types of values: integer values and Boolean values.

Integer values. Integer values are represented in AppleCore in the ordinary way for the Apple II:

1. Unsigned single-byte values are represented by treating the 8 bits, low to high, as the 8 digits of a binary number.

2. Multi-byte values are stored in little-endian order.

3. Signed values are stored in two's complement representation (i.e., by constructing the absolute value of the number, flipping the bits, and adding one).

Boolean values. The AppleCore language has no special Boolean type; as in C, ordinary integers serve as Boolean values. However, AppleCore differs from C in what values it treats as "true" and "false." In AppleCore, only the lowest-order bit is relevant in testing the Boolean value of an integer: integers whose low bit is 1 (i.e., odd integers) all count as "true," while integers whose low bit is 0 (i.e., even integers) all count as "false." This mechanism allows bitwise `NOT` to function as logical `NOT`, without any need for a separate logical negation operator, and that keeps the language simple. Also, the constants 1 and 0 can function as "true" and "false." These names and values are not built into the language, but one can easily declare constants `TRUE` and `FALSE` with the values 1 and 0.

4.2 Attribution

The first semantic check performed by the compiler is to match each identifier appearing in the source file with its corresponding definition; this step is called *attribution*. For purposes of these rules, a *definition* is a constant declaration, data declaration, variable declaration, function parameter declaration, or function declaration. A *use* is any identifier appearing in the source file not as the identifier of a definition.

The compiler matches definitions with uses in the following way:

1. At global scope (i.e., outside of any function body), for each use find the corresponding definition with the same identifier at global scope. If there is no corresponding definition, or if any two definitions use the same identifier, then report an error.
2. At function scope (i.e., inside a function body), for each use find the corresponding definition with the same identifier in the function scope, which consists of the global scope together with all function parameters and local variables declared in the function. If there is no corresponding definition, or if any two definitions in the function scope use the same identifier, then report an error.

Constant declarations must precede their uses in the source file. Otherwise, declarations need not precede their uses.

4.3 Constant-Value Expressions

A constant-value expression is one of the following:

1. A numeric constant.
2. An identifier corresponding to a constant declaration.
3. A binary operation expression, where each operand is a constant-value expression.
4. A negation (`-`) or `NOT` unary operation expression, where the operand is a constant-value expression.
5. A parentheses expression, where the expression inside the parentheses is a constant-value expression.

4.4 Expressions Appearing at Global Scope

An expression appearing at global scope (i.e., in a constant or data declaration, or as the initializer of a global variable declaration) must be one of the following:

1. A constant-value expression (Section 4.3).
2. An identifier corresponding to a data declaration (useful for data tables).
3. An identifier corresponding to a function declaration (useful for jump tables).

These rules are designed so that the compiler can do a single pass over all the expressions *expr* appearing at global scope, in the order in which they appear in the source file, and either compute the value of *expr* (in case 1) or generate an assembler label for *expr* (in cases 2 and 3). In case 3, the actual address associated with the assembler label may not be known until final assembly (Section 5.1).

4.5 Size and Signedness

Size and signedness of expressions. Every expression appearing in the source file is given a *size* and a *signedness*. The size and signedness of a constant-value expression (Section 4.3) come from the value produced by evaluating the expression as specified in Section 5.6, assuming that both operands have the maximum allowed size of 255. The size is the minimum number of bytes required to represent the value. The expression is signed if the value is less than zero, otherwise unsigned.

For an expression that is not a constant-value expression, size and signedness are computed as follows:

- *Identifiers.* The size and signedness of an identifier come from its definition (Section 4.2). (1) If the definition is a variable declaration or function parameter declaration, then the size is as given in the declaration, and the signedness is signed if `S` appears after the size, otherwise unsigned. (2) If the definition is a data declaration or function declaration, then the size is 2, and the signedness is unsigned.
- *Indexed expressions.* The size of an indexed expression is given by the *size* component of the expression. It is always unsigned.
- *Register expressions.* The size of a register expression is 1. It is unsigned.
- *Call expressions.* If the called expression (i.e., the expression before the first parenthesis) is anything other than an identifier corresponding to a function declaration, then the size is 0 and the signedness is unsigned. Otherwise the size and signedness come from the function declaration corresponding to the identifier: (1) If a *size* appears after the keyword `FN` in the definition, then that is the return size; otherwise the return size is 0. (2) If `S` appears after *size* then the signedness is signed, otherwise unsigned.
- *Binary operation expressions.* For comparison operations, the size is 1 byte unsigned. For shift operations, the size and signedness come from the left operand. For all other operations, the size is the maximum of the sizes of its operands. If either of the operands is signed, then the result is signed. Otherwise, the result is unsigned.
- *Unary operation expressions.* The size and signedness of a unary operation expression come from the operand, except that (1) an address operation `@` is two bytes unsigned; and (2) a negation operation is signed.
- *Parentheses expressions.* The size and signedness of a parentheses expression come from the enclosed expression.

Size and signedness requirements for expressions. The following requirements apply to the size and signedness of all expressions, whether or not they are constant-value expressions:

- *Indexed expressions.* Both the base expression (before the first bracket) and the index expression (immediately after the first bracket) must have size 1 or 2.
- *Call expressions.* (1) The called expression must be an expression of size 1 or 2. (2) Each argument expression must have nonzero size.
- *Binary operation expressions.* For shift operations, the right-hand expression must be 1 byte unsigned.

Set statements. The right-hand expression of a set statement must have nonzero size.

Return statements. (1) No return statement may contain an expression of zero size. (2) If a function has nonzero return size, and the function has a body, then (a) the body must end with a return statement containing an expression, and (b) every return statement appearing in the body must contain an expression. (3) If a function has return size 0, then no return statement in the function body may contain an expression.

Function frame sizes. The *frame size* of a function declaration that contains a body equals the sum of the sizes of all the function parameters and local variables. No function may have a frame size of greater than 255 bytes.

4.6 Number of Function Arguments

At every call expression where the called expression is an identifier corresponding to a function declaration, the number of arguments must match the number of parameters given in the declaration.

4.7 Semantic LValues

Definition of semantic lvalue. A *semantic lvalue* is an expression that (1) is an identifier, register expression, or indexed expression; and (2) if it is an identifier, then it corresponds to a variable or function parameter declaration.

Requirement of semantic lvalues. (1) A semantic lvalue is required in the following places: (a) on the left-hand side of a set statement; (b) as the expression of an increment or decrement statement; and (c) as the operand of an address (`@`) unary operator. If an expression that is not a semantic lvalue appears in any of these places, then a compile-time error results. (2) A compile-time error results if a register expression appears as the operand of an address (`@`) operator.

5 Code Generation

5.1 Source Files

The AppleCore language is designed to support separate compilation. To do that, the compiler translates source files to assembly files in one of two modes: *top-level mode* and *include mode*. The mode must be specified at the time the source file is translated to assembly, usually as a compiler option.

Top-level mode. In top-level mode, the compiler translates the source file as follows:

1. Issue some code (or an assembler directive to include the code) that sets up the program stack (Section 5.3).
2. Issue code to transfer control to the first function with a body that appears in the file. There must be at least one such function, and it is a compile-time error if not.
3. Translate all global declarations in the source file as described in Section 5.2, in the order in which they appear in the source file. Include declarations may appear anywhere in the file, and they are translated to directives to include the corresponding assembly files.
4. Issue assembler directives to include the AppleCore support code (for example, the code needed to do arithmetic operations).

Include mode. In include mode, the compiler just translates the declarations in the source file as described in Section 5.2. In include mode, the source program may not contain any include directives, and it is a compile-time error for one to appear.

Final assembly and symbol resolution. A user program consists of exactly one file translated in top-level mode and zero or more files translated in include mode. The final step in compilation is to ask the assembler to assemble the file translated in top-level mode, which includes all the other files for assembly. This assembly also causes name resolution for externally defined functions. It is the user's responsibility to ensure that externally-defined names are both available and uniquely defined; the compiler cannot check this. If a symbol used in one of the assembled files is not defined in any of the files (or if any symbol is multiply defined), then an assembler error will result.

5.2 Global Declarations

Global declarations are translated as stated below. To the extent that translation produces actual bytes of machine code, data, or reserved storage, the bytes must be laid out in the order in which the translated constructs are encountered in the source file.

Constant declarations. The compiler (1) evaluates the expression of the constant declaration to a numeric constant according to the rules in Section 5.6 and (2) associates this value with the name given in the constant declaration. The

compilation must support the use of the same identifier for different constant declarations in different files that are part of the same final assembly (Section 5.1).

Data declarations. (1) If an expression appears before the semicolon, then the compiler evaluates the expression to a numeric constant according to the rules in Section 5.6 and generates code to store the bytes of the constant in little-endian order. (2) If a string constant appears before the semicolon, then the compiler generates assembly code to store the bytes of the string constant in left-to-right order. (a) If no backslash \ follows the string constant, then the compiler adds a terminating NUL ($00) character after the last byte of string data. (b) If a backslash follows the string constant, then no NUL character is added. (3) In any case, if an identifier appears after the keyword `DATA`, then the compiler associates the name with the lowest address in which data is stored.

Global variable declarations. The compiler sets aside a number of bytes of storage equal to the declared size of the variable and associates the identifier appearing after `VAR` with the lowest address of this storage. If an expression appears before the semicolon, then the compiler evaluates the expression to a numeric constant according to the rules stated in Section 5.6 and issues assembly code to fill in the storage set aside for the variable with the bytes of the numeric constant, in little-endian order, using size adjustment (if necessary) as stated in Section 5.8.

Function declarations. (1) A function declaration with no body does not correspond to any generated code; its signature is used only for checking call expressions that invoke the function. (2) A function declaration with a body causes code to be generated as stated in Section 5.4.

Include declarations. In top-level mode (see Section 5.1), the compiler translates an include declaration into an assembler directive to include the file named in the string constant. In include mode, the presence of an include declaration in the source file causes an error.

5.3 Program Stack

This specification requires that the implementation provide three logical stacks for use at runtime, collectively referred to as the *program stack*:

1. A C-style *call stack*, such that a new frame appears at the top of the stack for each function call, and the frame is popped at the end of the call. This stack is used in generating function call code, as specified in Section 5.4.
2. An *expression stack* for use during expression evaluation as specified in Section 5.6.
3. An *allocation stack* that supports an `ALLOCATE` library function that works as follows. (1) `ALLOCATE` is only ever called when no temporary expression result is on the expression stack; otherwise the results are undefined. (2) When `ALLOCATE` is called, it is passed a two-byte size value s. (3) (a) If there is enough space on the stack, then `ALLOCATE` returns a pointer to s bytes on the stack. (b) If there are not s bytes remaining on the stack, then the results are undefined. (4) The s bytes pointed to by the result of the `ALLOCATE` call remain valid (i.e., are not modified except by user-written code) during the lifetime of the call stack frame for the function in which the `ALLOCATE` call occurred. (5) At the end of the function in which the `ALLOCATE` call occurred, the memory is deallocated and may be reused as program stack storage.

The compiler may provide this stack functionality in any way that is feasible. Typically the same physical stack would be used for all three logical stacks (and the logical stacks have been specified so that this is possible), but this is not a requirement. Also, typically the generated code would maintain its own two-byte stack pointer, because the native Apple II stack located at addresses $200 through $2FF is too small to serve as the program stack for this language. However, this specification does not mandate any particular stack size; the actual available stack size depends on both the compiler implementation and the program memory requirements (for example, whether the graphics and/or language card areas are available for program heap and stack data).

5.4 Function Bodies

The AppleCore language definition assumes a C-style function call stack as specified in Section 5.3. For each function declaration that has a body, the compiler does the following:

- Generate code to store the callee's frame pointer on the call stack.
- Generate code to reserve slots on the call stack for the function parameters, local variables, and saved registers. The saved registers are all the registers appearing in register expressions in the function body.
- For each local variable declaration that includes an initializer expression, generate code to evaluate the expression according to the rules in Section 5.6 and assign the resulting value into the stack slot for the variable, using size adjustment (if necessary) as stated in Section 5.8.
- For each statement of the function body, generate code as stated in Section 5.5. The statements of a function body are executed in sequence, in the order in which they appear in the program text.
- If the last statement executed in the function is not a return statement, then generate code to pop the frame off the stack and restore the parent frame.

This calling convention is designed so that from the point of view of the caller, a call to an AppleCore function that takes no arguments and returns no result is indistinguishable from a `JSR` to a non-AppleCore function (for example, in the System Monitor). That way, non-AppleCore assembly language functions can be called in the same way as AppleCore functions. In particular, if `V` is a variable, then a call expression `V()` has the same meaning regardless of what kind of function address is stored in `V` (which is not generally known at compile time). Also, a source file with a function declaration `FN FOO();` can be assembled with any assembly file containing a function with label `FOO`, regardless of whether the code at label `FOO` uses the AppleCore conventions for managing the stack.

5.5 Statements

The compiler generates code to do the following for each program statement.

If statements. Evaluate the conditional expression (Section 5.6), pop the result off the expression stack, and mask off all but the lowest bit of the result to get the corresponding Boolean value (Section 4.1). If the Boolean value of the result is 1 (true), then execute the true part. If the Boolean value of the result is 0 (false), then either do nothing (if there is no optional `ELSE` clause) or execute the statement in the `ELSE` clause (if one is provided).

While statements. Evaluate the conditional expression (Section 5.6), pop the result off the expression stack, and mask off all but the lowest bit of the result to get the corresponding Boolean value (Section 4.1). If the Boolean value of the result is 1 (true), then execute the body of the loop and repeat the whole process. If the Boolean value of the result is 0 (false), then do nothing.

Set statements. (1) Evaluate the right-hand side expression, adjusting the size if necessary (Section 5.8) to conform to the size of the left-hand side expression. (2) The left-hand side expression must be a semantic lvalue (Section 4.7). Compute its address as specified in Section 5.7. (3) Copy the result of step (1) into memory starting at the address computed in step (2).

Notice that the right-hand side of the set statement is evaluated first; this is for implementation efficiency reasons. In particular, the expression `SET X[0,1]=X+1` is evaluated by *first* adding 1 to `X` and *then* computing the destination address, so the effect is to store the original value of `X` plus 1 at the location given by the original value of `X` plus 1.

Call statements. Evaluate the expression contained in the statement (Section 5.6). If it produces a result of nonzero size, pop the result off the stack.

Increment statements. The operand expression must be a semantic lvalue. (Section 4.7). Compute its address as stated in Section 5.7. Add one to the value stored at the address and store the result in place.

Decrement statements. The operand expression must be a semantic lvalue. (Section 4.7). Compute its address as stated in Section 5.7. Subtract one from the value stored at the address and store the result in place.

Return statements. (1) If the statement contains an expression, then evaluate the expression (Section 5.6) and ensure that it appears at the top of the expression stack (Section 5.3) on function return. (2) Pop the current frame off the call stack and restore the parent frame.

Block statements. Execute each statement of the block in the order in which it appears in the program text.

5.6 Expressions

We define the semantics of expressions using an expression stack, as specified in Section 5.3. The compiler generates code to do the following for each program expression.

Identifiers. (1) If the identifier is a semantic lvalue (Section 4.7), then compute its address as specified in Section 5.7 and get a number of bytes equal to the size of the right-hand side (Section 4.5), starting at that address. Push those bytes on the stack, using size adjustment if necessary (Section 5.8) to generate a result that has a number of bytes equal to the left-hand expression size. (2) If the identifier corresponds to a numeric constant declaration, then push the bytes of the value associated with the declaration (Section 5.2) on the stack in little-endian order. (3) If the identifier corresponds to a data declaration, then push the two-byte address associated with the identifier (Section 5.2) on the stack in little-endian order. (4) If the identifier corresponds to a function declaration, then push the two-byte address corresponding to the first byte of the function body (Section 5.4) on the stack in little-endian order. (If the function body is defined in a different file, this can be done by using the label associated with the function body; at assembly time, the label definition will be available.)

Numeric constants. Push the bytes of the value associated with the declaration (Section 5.2) on the stack in little-endian order.

Indexed expressions. (1) Compute the address of the indexed expression as stated in Section 5.7. (2) Get a number of bytes equal to the size specified in the expression, starting at that address, and push them on the stack.

Register expressions. Push the one-byte value stored in the function call stack slot for that register (Section 5.4) on the expression stack.

Call expressions. The procedure depends on the kind of the called expression.

(1) Declared functions: If the called expression is an identifier corresponding to a function declaration, then do the following. (a) Evaluate each argument and store it to its slot on the stack in the new frame, adjusting the argument sizes (Section 5.8) as necessary to them to the parameter sizes specified in the function definition. (b) Adjust the stack pointer to point to the base of the new frame. (c) Restore all the saved registers (Section 5.4) to the values saved in their slots on the stack. (d) Do a `JSR` to the code for the function body (Section 5.4). (e) Save all the saved registers to their slots.

(2) Compile-time constants: If the called expression evaluates to a compile-time constant, then (a) restore the registers as in (1)(c), (b) `JSR` to the constant address, and (c) save the registers as in (1)(e).

(3) Callee address unknown: Otherwise the callee address is not known at compile time. In this case proceed as in case 2, but in step (2)(b) store the value to zero-page memory and do a `JSR` to an indirect `JMP` to the address.

The rules are designed so that in all cases the last value written into the stack slot for each saved register prior to the call is guaranteed to be in the corresponding 6502 register just before the `JSR` to the called function. Thus, except for the `S` register, all saved registers are guaranteed to have their saved values at the point where control enters the callee function. Because of the `JSR`, the `S` register's value will be increased by 2. Similarly, all stack slots corresponding to

saved registers other than the S register will contain the values that the corresponding registers had on exit from the function; the S register's value will be off by 2.

Binary operation expressions. *Shift operations (<< and >>).* Evaluate the left-hand side, evaluate the right-hand side, pop the right-hand side, and shift the left-hand side value (which is the result on the top of the stack) by a number of bytes equal to the right-hand side value. In the case of a right shift, the shift is signed (arithmetic) if the left-hand side expression is signed; otherwise it is unsigned (logical).

Arithmetic operations (, /, +, and −).* Evaluate the left and right expressions, adjust their size if necessary (Section 5.8) to make their sizes equal to the size of the entire expression, pop the results, perform the operation, and push the result. In the case of multiplication and division, the operation is signed if the entire expression is signed, otherwise unsigned. Because AppleCore uses two's complement representation for negative numbers (Section 4.1), there is no distinction between signed and unsigned addition and subtraction. The division operation produces the integer quotient. All operations are done without regard to overflow (i.e., if overflow occurs, they produce the low-order bits, up to the size of the expression).

Comparison operations (=, >, <, >=, and <=). Evaluate the left and right expressions, adjust their size if necessary (Section 5.8) to make their sizes equal to the size of the entire expression, pop the results, perform the operation, and push the result. The result is a single-byte value such that 1 represents true and 0 represents false. In the case of comparisons other than equality, the operation is signed if the entire expression is signed, otherwise unsigned.

*Bitwise operations (*AND, OR, and XOR*).* Evaluate the left and right expressions, adjust their size if necessary (Section 5.8) to make their sizes equal to the size of the entire expression, pop the results, perform the operation, and push the result.

Unary operation expressions. *Dereference (@).* The operand expression must be a semantic lvalue (Section 4.7). Compute and push its address as stated in Section 5.7.

*Arithmetic negation (−) and bitwise not (*NOT*).* Evaluate the operand expression and perform the operation on it. Negation is two's complement negation (i.e., NOT plus 1).

Parentheses expressions. Evaluate the expression inside the parentheses.

5.7 Address Computation for Semantic LValues

Computation of the address of a semantic lvalue *expr* works as follows:

1. If *expr* is an identifier corresponding to a global variable, then its address is the lowest address of the global storage assigned to that variable (Section 5.2).
2. If *expr* is an identifier corresponding to a function parameter or local variable, then its address is the lowest address of the function call stack slot assigned to that parameter or variable (Section 5.4).
3. If *expr* is an indexed expression, then its address is computed by evaluating the base expression, evaluating the offset expression, popping and adding the results, and adjusting the size of the result to 2 if necessary (Section 5.8).
4. If *expr* is a register expression, then its address is the address of the function call stack slot assigned to that register (Section 5.4).

5.8 Size Adjustment

Size adjustment applies whenever a result is assigned into or out of a semantic lvalue and the sizes of the source and target values do not match. The following rules apply to size adjustment:

1. If the source value size is greater than the target value size, then high-order bytes of the source value are *truncated*: only the low-order bytes of the source value are stored, up to the number of bytes required by the target value.

2. If the source value size is smaller than the target value size, and the source value is unsigned, then the source value is *zero extended*: the bytes of the source value form the low-order bytes of the target value, and zeros are used to fill the rest of the bytes of the target value.

3. If the source value size is smaller than the target value size, and the source value is signed, then the source value is *sign extended*: the bytes of the source value form the low-order bytes of the target value, and the rest of the bytes of the target value are filed with either all 0 bits (if the high-order bit of the source value is 0) or all 1 bits (if the high-order bit of the source value is 1).