Language Reference Manual - simpliCity

Course: COMS S4115
Professor: Dr. Stephen Edwards
TA: Graham Gobieski
Date: July 20, 2016

Group members
Rui Gu - rg2970
Adam Hadar - anh2130
Zachary Moffitt - znm2104
Suzanna Schmeelk - ss4648

Table of Contents

Introduction

The simpliCity language is a simplified version of C which was developed by Kernighan and Ritchie. The simpliCity language contains a subset of C grammar but with a strict type system. The language uses LLVM as the backend to produce bytecode.

The simpliCity language operates best with programs that need to be Turing complete and can be defined using strict type casting and only stack-based memory management, which decreases runtime errors. The C language domain is for number crunching and embedded systems. SimpliCity programs will operate in the same domain as C with the exception that our compiler supports only a limited features of C.

Lexical Conventions

1. Identifiers

An identifier is any sequence of alphanumeric characters, where the first character must be alphabetic. The `_` character is the only non-alphanumeric symbol accepted in an identifier, and it is read as an alphabetic character.

2. Comments

Comments are introduced with the `/*` character string, and terminate with the `*/` character string. All characters within these indicators are ignored. Comments do not nest.

3. Whitespace

Whitespace is basically ignored by the compiler; any combination of whitespace characters will be interpreted as one whitespace character.

4. Keywords

The following identifier are reserved for various uses, and cannot be used in any other way than how they are specified (later in this manual):

- `int`
- `chr`
- `sci`
- `txt`
- `bool`
- `true`
- `false`
- `struct`
- `start`
- `return`
- `break`
- `continue`
- `if`
- `else`
- `while`
- `for`

- `print`
- `scan`

Constants

1. Integer constants

   An integer constant is a sequence of digits. It is taken to be a decimal number.

2. Character constants

   A character constant is 1 or 2 characters enclosed in single quotes, `‘` and `’`. To represent a single quote character as a character constant, it must be preceded by a backslash, e.g. `\’`. The backslash character is used as an escape for several other special character constants, as shown in this table:

Backslash
Single quote
Double quote
New line
End of string / null byte
	`\\`
	`\’`
	`\”`
	`\n`
	`\0`

3. Floating constants

   A floating constant is made up of an integer part (written the same as an integer constant), a decimal point (the `.` character), and a fractional part (written the same as an integer constant). To be recognized as a floating constant, it needs to have at least the decimal point, and either the integer part or the fractional part. A floating constant is single-precision.

4. Strings

   A string is a set of characters surrounded by double quotes, `“` and `”`. A string is considered in the back end as an array of characters, which is held in memory as a contiguous block of data. To represent the double-quote character within a string, it must be preceded with the escape character as specified for character constants, e.g. `\”`. The other special characters specified there should be written in the same method.

Objects, types, and conversion

1. Fundamental types

   SimpliCty supports four fundamental types of objects - integers, characters, single-precision floating-point numbers, and booleans:

- characters (from here on labelled `chr`), are representative of the ASCII character set; they are the rightmost 7 bits of a single byte. They can be manipulated as if they are 1 byte 2's complement numbers (the MSB is a 0).
- integers (`int`) are 16 bit (2 byte) 2's complement numbers.
- single-precision numbers (`sci`) are 32 bit (4 byte) numbers represented with 24 bits of precision, 8 bits for an exponent, and 1 bit for a sign.
- booleans (`bool`) are 8 bit (1 byte) representations of true or false statements. The reserved keywords `true` and `false` are its two possible values.

2. Derived types

There are three types which can be constructed from the fundamental types:

- arrays - a set of objects that are all the same type. A string is a type of array (an array of `chr`) which has the special label `txt`.
- structures - a set of objects that may not be all the same type
- functions - a subroutine that returns an object of a specific type

These derived types are generally recursive. There can be an array of arrays, an array of structures, a structure of arrays, a structures of structures, and a function can return an array or a structure. Functions cannot return functions.

3. 'lvalues'

'lvalues' are expressions that refer to objects. An identifier is an lvalue. When the code is being parsed, expressions like `a = b` or `a = 3` will interpret `a` as being an lvalue. `b` is also an lvalue in this example.

4. Conversions

SimpliCty is a strongly typed language, and so it does no type conversion whatsoever. Any attempt to do so natively will result in a compiler error. To convert one type to another one must use an external library.

Expressions

Precedence of expressions is represented in this manual in the order of their section numbers. For example, all expressions from subsection 3 (unary operators) will always take precedence over any expression in subsection 4 (multiplicative operators), or subsection 5 (additive operators). It will always be ignored before expressions from subsection 2 (primary expressions), however.
Expressions defined within the same subsection have undefined precedence over each other. The compiler will arbitrarily compute them, in whatever order.

1. Syntax notation

Syntactic categories are indicated by text in *the font Cambria, and in italics*. Literal words or symbols are written in `Courier New, with a light gray background`. The subscript characters $_{opt}$ mark a category that is optional.

2. Primary expressions

Primary expressions group left-to-right.

a. *identifier*
An identifier is a primary expression, but only if it has been properly declared. Upon declaration, its type must be specified. However, if the type of the identifier is "array of type T", then the value of the identifier expression is an internal pointer to the first object in the array. The user will not be able to manipulate the address directly, as pointers are not available to be manipulated.

b. *constant*
Decimal, character, boolean, or floating point constants are all primary expressions. It's type is `int` for decimal integers, `chr` for character, `sci` for floating point, and `bool` for boolean.

c. *string*
A string is a primary expression, whose type is 'array of `chr`'.

d. `(` *expression* `)`
Parenthesized expressions are primary expressions who would be evaluated identically to the same expression without parentheses. This is useful to avoid ambiguity in writing expressions.

e. *primary-lvalue* `[` *expression* `]`
This expression defines a reference to a specific value within an array. It is a valid primary expression when the left expression resolves to a variable of type array, and when the right expression (within brackets) resolves to an `int` which is not larger than the size of the array.

f. *primary-lvalue* `->` *member-of-structure*
This expression is a valid primary expression when the lvalue is referring to a structure, and the member of the structure referred exists within that structure.

3. Unary operators

Unary operations group right-to-left.

a. `-` *expression*
The "mathematical negation" expression results in the negative of the given expression of the same type. It only operates on `int`, `chr`, and `sci`. The output of this operation is the same type as the inner expression.

b. `!` *expression*

The "logical negation" expression results in `true` if the expression resolves to 0, and `false` if the expression resolves to a non-zero value. It only operates on `bool`. The output of this operation is a `bool`.

c. `++` *lvalue-expression*
   The "increment" expression increments by 1 the object referred to by the lvalue, as long as it is of type `int` or `chr`. The final expression returns this incremented value, of the same type as the lvalue object.
d. `--` *lvalue-expression*
   The "decrement" expression operates similarly to "increment" but it subtracts 1.

4. Multiplicative operators

Both multiplicative operators and the following subsection (additive operators) group left-to-right, in order to emulate math in common usage.

a. *expression* `*` *expression*
   The `*` operator expresses multiplication. Both expressions on either side must be of the same valid type, `int` or `sci`. The result of this operation is the same type as its two operands.
b. *expression* `/` *expression*
   The `/` operator expresses division. Its type requirements are the same as multiplication. Note that on `int/int` division, this operation throws away the remainder to keep its output `int`.
c. *expression* `%` *expression*
   The `%` operator expresses the modulo operation. Its type requirements are the same as the other multiplicative operators.

5. Additive operators

a. *expression* `+` *expression*
   The `+` operator expresses addition. Its type requirements are the same as the multiplicative operators; both operands must be of the same type (both `int`, both `chr`, or both `sci`), and it returns a value of the same type as its operands.
b. *expression* `-` *expression*
   The `-` operator expresses subtraction. Its type requirements are the same as addition and the multiplicative operators.

6. Relational operators

Relational operators (as well as the two following section 'Equality' and 'Comparison') resolve its two operands into an output that is a `bool`. These three sections all group left-to-right.

a. *expression* `<` *expression*
b. *expression* `<=` *expression*
c. *expression* `>=` *expression*
d. *expression* `>` *expression*

These operators each evaluate to `true` if the relation is true, and `false` otherwise. Both operands must be of the same type. Types `int`, `chr`, `sci`, and `bool` are accepted. The output of these operations is a `bool`.

7. Equality operators
    a. *expression* `==` *expression*
    b. *expression* `!=` *expression*
    These operators are equivalent in practice to the relational operators, but they always have lower precedence.

8. Comparison operators

SimpliCty does not support bitwise operations. As such, the comparison operators do not need to be a double character.

a. *expression* `&` *expression*
    Returns `true` if both operands are nonzero, otherwise `false`. If the first expression evaluates to `false`, the second expression is not evaluated. All primitive types are accepted as operands, but a `bool` is outputted.
b. *expression* `|` *expression*
    Returns `true` if both or either operands are nonzero, otherwise `false`. If the first expression evaluates to `true`, the second expression is not evaluated. All primitive types are accepted as operands, but a `bool` is outputted.

9. Assignment operators

Assignment operations group right-to-left. They all require an lvalue as the left operand. The value returned is the value that is placed in the object referred to by the lvalue. The object referred to by the lvalue must be declared before it can be assigned.

a. *lvalue* `=` *expression*
    Both the object referred to by the lvalue, and the evaluated expression, must have the same type.
b. *lvalue* `+=` *expression*
c. *lvalue* `-=` *expression*
d. *lvalue* `*=` *expression*
e. *lvalue* `/=` *expression*
    The behavior of these four operations resolve to (for example `+=`) `lvalue = lvalue + expression`. The mathematical operation detailed in the symbol evaluates that operation on the object referred to by the lvalue and the expression, and then that value is assigned to the lvalue. The types of of the lvalue and the expression must match, as detailed in section 4 and 5.

10. Concatenation operator
    a. *expression1* `.` *expression2*

The concatenation operator `.` takes as operands two expressions that resolve to arrays of the same type, and outputs a single array of the same type. For example, if *expression1* was declared as `int a1[5]`, where `a1[5] = {0;1;2;3;4}` and *expression2* was declared as `int a2[3]`, where `a2[3] = {5;6;7};`, then the operation `a1.a2` would resolve to the array `{0;1;2;3;4;5;6;7}`. If the first expression is made up of elements of type 'array of `bool`', then the second expression must also be made up of elements of type 'array of `bool`'.

Declarations

Declarations are used within functions to declare instances of designated type. Declarations have the following form

declaration:

decl-specifier declarator-list$_{opt}$ `;`

The declarator list contains all the identifiers waiting to be declared. For decl-specifier, only one specifier is allowed for each declaration.

decl-specifiers:

type-specifier

Note that a variable cannot be declared and assigned a value to in the same statement.

1. Type specifiers

The only types that can be specified are the primitive types, the special character array `txt`, and data structures.

type-specifier:

`int`
`chr`
`sci`
`bool`
`txt`
struct-specifier

2. Declarators

A declarator list is a list of declarators

declarator-list:

declarator `,` *declarator-list*
declarator

Where a declarator is a variable name, and optionally a definition of an array. A declarator list must exist for all types, except for data structures, where they are optional.

declarator:

identifier
declarator `[` *constant* `]`
`(` *declarator* `)`

Together with the associated type specifiers, each declarator yields an instance of the indicated type. A declarator with the form of *declarator* `[` *constant* `]` indicates we are declaring an instance of *array*, with size *constant*. If the type-specifier was `txt`, the size of the array does not need to be specified. An array may be constructed from one of the primitive types, from a structure, or from another array (to generate a multidimensional array).

3. Structure Declarations

A data structure specifies a new composite type, which is composed of one or more primitive types, array, or other previously specified data structures.

struct-specifier:

`struct` `{` *type-decl-list* `}`

`struct` *identifier* `{` *type-decl-list* `}`

The *type-decl-list* is a sequence of type declarations for the members of the structure. *type-declaration* is just normal declaration with *type-specifier* and *declarator*.

As stated above, the declarator list is optional for structure declaration. A structure declaration can be specified for one or multiple variables in one statement or, alternatively, just the type can be declared, with variables of that type declared later.

Note that the first instance of the struct specifier, `struct` `{` *type-decl-list* `}`, requires a declarator list, while the second does not.

SimpliCty does not allow self-referential structures. The declaration for structures is otherwise similar to the way one declares a variable. However, the *declarations* within the *type-decl-list* should always have names as well.

type-decl-list:

type-declaration

type-declaration type-decl-list

Statements

1. Expression statement

An expression statement has the form:

expression `;`

Expression statements are assignments or function calls.

2. Compound statement

Write multiple expression statements, that will be evaluated one after the other, like this:

`{` *stat-decl-list* `}`

stat-decl-list will be more clearly defined in the section “Program definitions” but for now it will be defined as:

stat-decl-list:

statement statement-list

statement

3. Conditional statement

The two types of conditional statements are:

`if` `(` *expression* `)` *statement*

`if` `(` *expression* `)` *statement* `else` *statement*

If the expression surrounded by parentheses evaluates to `true` (it must be a `bool`), then the first statement is evaluated.

In the second instance of a conditional, the second statement is evaluated if the expression evaluated to `false`. Never are both statements evaluated.

4. While statement

The while statement is expressed as such:

`while` `(` *expression* `)` *statement*

The expression is evaluated - if it evaluates to a `true` then the statement is evaluated. The expression must be a `bool`. Control flow then jumps back to the expression and re-evaluated. This process is repeated until the expression evaluates to `false`.

5. For statement

The for statement is expressed thusly:

`for` `(` *expression1* `;` *expression2* `;` *expression3* `)` *statement*

But this statement is equivalent to:

```
expression1 ;
while ( expression2 )
{
        statement
        expression3 ;
}
```

There must be an expression in each of the three positions.

6. Break statement

The statement

`break` `;`

Makes the latest `while` or `for` statement terminate prematurely. Control flow moves to the statement following the terminated statement.

7. Continue statement

The statement

`continue` `;`

Can only be used on `while` or `for` statements, in order to prematurely jump back to the evaluation of the potentially `false` expression.

8. Return statement

The statements

`return ;`

`return (` *expression* `) ;`

Move control flow back to the caller of the function within which these statements have been expressed. In the first type of statement no value is returned. In the second, the expression must evaluate to the type specified in the function definition.

9. Print statement

The statement

`print (` *expression* `) ;`

Prints the expression to the command line. The expression must resolve to a `txt`.

10. Scan statement

The statement

`scan ( ) ;`

Pulls the next character types by the user in the command line until the a new line character or null terminator is received. This statement outputs a `chr`.

Program definitions

A simpliCty program consists of series of external definitions. An external definition is either a function definition or a global data structure definition.

program:

external-definition program

external-definition

external-definition:

function-definition

struct-definition

1. Function definition

A function definition is defined as

function-definition:

type-specifier function-declarator function-body

Where the type specifier details what the function returns, the function body details the statements that will be evaluated when the function is called, and the function declaration

function-declarator:

declarator `(` *parameter-list*$_{opt}$ `)`

`start ( )`

Details the name of the function (*declarator*), and the series of parameters that are passed to the function as inputs. The special case of the function declaration, with the keyword `start`, is used to define the entry point for control flow. It is equivalent to the "main" function keyword in the regular C Language. `start` does not have any argument inputs.

A parameter list is defined as

parameter-list:

identifier `,` *parameter-list*

identifier

Since the parameter list is optional, a function does not necessarily need any inputs.

A function body has the form

function-body:

type-decl-list function-statement

function-body:

`{` *stat-decl-list* `}`

stat-decl-list:

statement-declaration stat-decl-list

statement-declaration

statement-declaration:

statement

declaration

stat-decl-list was earlier defined as only a list of statements. Its full definition allows it to be a list, of arbitrary length, of either statements or declarations. At least one of the statements in a function must be a "return" statement, as the function needs to at some point return control flow to its parent function (or terminate the program).

Once the specially defined function `start` "returns", a program terminates.

2. Structure definition

A struct definition is defined as

struct-definition:

struct-specifier `;`

A structure is defined here in the same manner as previously defined in the declarations section, although it does not allow for the declaration of variables of this new data structure type. A structure may be defined in this scope when it needs to be defined as an input or a return value for one or multiple functions. A structure defined within a function will be undefined (as out of scope) once the function returns control flow.

Scope and Preprocessor rules

Variables declared within a function are undefined when the function returns. Structures defined within a function become undefined when the function returns. If the user wants a structure to be available to more than one function, it must be defined at the level of other functions (as specified in the previous section, “Program definitions”).

1. File inclusion

The source text of a simpliCty program does not need to be all in one file. There are special preprocessor rules that allow the importing of the contents of an external file at the start of the file, before compilation.

`#include " ` *filename* `"`

Where the *filename* is a relative path to the external file.

2. Token replacement

Any string of characters can within a source file can be replaced with another set of characters using the token replacement rule:

`#define` *identifier token-string*

The identifier is not the string to be replaced, and the token-string is the new set of characters to replace it.

The compiler is not responsible for any errors that result from this replacement. The programmer must be attentive.

Appendix - Context Free Grammar

1. Expressions

expression:
- *primary*
- `-` *expression*
- `!` *expression*
- `++` *lvalue*
- `--` *lvalue*

primary:
- *identifier*
- *constant*
- *string*
- `(` *expression* `)`
- *primary* `[` *expression* `]`
- *primary* `->` *identifier*

lvalue:
- *identifier*
- *primary* `[` *expression* `]`
- *lvalue* `->` *identifier*
- `(` *lvalue* `)`

expression-list:
- *expression* `,` *expression-list*
- *expression*

The primary expression operators

`()` `[]` `->`

Have highest priority and group left-to-right. The unary operators

`-` `!` `++` `--`

Have priority below the primary operators but above any binary operators, and group right-to-left. The binary operators

`*` `/` `%`

`+` `-`

`<` `<=` `>=` `>`

`==` `!=`

`&` `|`

Group left or right, with priority decreasing. Assignment operators

`=` `+=` `-=` `*=` `/=`

Have the next priority, and group right-to-left. The concatenation operation

`.`

Has the lowest priority, and groups left-to-right.

2. Declarations

declaration:

type-specifier declarator-list$_{opt}$ `;`

type-specifier:

`int`
`chr`
`sci`
`bool`
`txt`
struct-specifier

struct-specifier:

`struct` `{` *type-decl-list* `}`
`struct` *identifier* `{` *type-decl-list* `}`

declarator-list:

declarator `,` *declarator-list*
declarator

declarator:

identifier
declarator `[` *constant* `]`
`(` *declarator* `)`

type-decl-list:

declaration type-decl-list
declaration

3. Statements

statement:

expression `;`
`{` *stat-decl-list* `}`
`if` `(` *expression* `)` *statement*
`if` `(` *expression* `)` *statement* `else` *statement*
`while` `(` *expression* `)` *statement*
`for` `(` *expression* `;` *expression* `;` *expression* `)` *statement*
`break` `;`
`continue` `;`
`return` `;`
`return` `(` *expression* `)` `;`
`print` `(` *expression* `)` `;`
`scan` `;`

4. Function Definitions

program:

external-definition program

external-definition

external-definition:

function-definition

struct-definition

function-definition:

type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:

declarator `(` *parameter-list*$_{opt}$ `)`

`start` `()`

parameter-list:

identifier `,` *parameter-list*

identifier

function-body:

`{` *stat-decl-list* `}`

stat-decl-list:

statement-declaration stat-decl-list

statement-declaration

statement-declaration:

statement

declaration

struct-definition:

struct-specifier `;`

5. Preprocessor

`#` `include` `"` *filename* `"`

`#` `define` *identifier token-string*