Language Documentation

Ruben Saunders

September 2024

Contents

1 Overview

The compiler takes one or more source files and produces single, linked assembly file.

1.1 Command-Line Interface

The compiler executable is called as follows:

```
$ ./compiler <input_file> -o <output_file> [flags]
```

The output file is provided after the `-o` flag. The following optional flags are available:

- `-d`: enables debug mode. In this mode, detailed results from each step are output to the console.
- `-l <file>`: for each file, writes lexed source to `file` as XML.
- `-p <file>`: for each file, writes parsed contents to `file` as XML.

1.2 Process Flow

The process flow bears resemblance to the assembler, and hence will be summarised in less detail.

1. For each input file:
   (a) Open the file.
   (b) Pre-process the file.
   (c) Parse the file, construct an AST.
   (d) Create symbol table.
2. Link referenced but undefined symbols using all symbol tables.
3. Compile components to assembly.
4. Write all components to assembly output.

2 General Syntax

Each file is parsed into tokens, which are then grouped in recognisable sequences. Tokens are grouped into lines; lines are separated by: a newline, unless the previous token was an operator or the parser is in a bracketed group '`(...)`'; or a semicolon '`;`'.
Each file has a global scope, which may only consist of global variables, functions, and data definitions. Below are outlined key points about syntax; specifics will be documented later.

- Single-line commends have the form `// ...` and last until a newline character is encountered.
- Multi-line commands have the form `/* ... */`.
- Identifiers start with a lowercase character which may then be followed by any number of characters, numbers, or underscores. That is, `[a-z][a-zA-Z0-9_]*`.
- Datatype identifiers start with an uppercase character which may then be followed by any number of characters, numbers, or underscores. That is, `[A-Z][a-zA-Z0-9_]*`.

3 Language Options

These options are provided to configure the language, enforce syntax, or modify reporting. They are listed internally, but currently cannot be changed without editing `src/LanguageOptions.cpp`.
Some options take the form of a reporting level. The possible values are:

- `-1` – hidden; disables the reporting.
- `0` – notice.
- `1` – warning.
- `2` – error; compilation is halted.

3.1 `allow_alter_entry`

Default: `true`
Enables use of the `entry` keyword to alter the program's entry point.

3.2 `allow_shadowing`

Default: `true`
Enables variable shadowing, wherein existing variables may be re-defined in the same scope.

```
decl a: byte
// ...
decl a: word
```

3.3 `must_declare_functions`

Default: `false`
Enforces the declaration of a function signature prior to its definition. That is, all `func ...` must be preceded by a matching `decl func ...`.

```
decl func add(int, int) -> int
// ...
func add(x: int, y: int) -> int { ... }
```

4 Types

Each variable has a type, indicating tha form, size, and purpose of some data.

Type Coercion This refers to how a value takes on a type given context, and occurs implicitly and only if necessary. For example, the literal 42 could take on different types depending on the variable's type. Another example would be adding two integers of different types; the smaller is coerced into the larger type.

Type Casting This is the explicit conversion of data between two types. Examples would be: down-sizing an integer value, changing a pointer type. This is done by preceding a variable or value by a bracketed type. For example,

```
decl pi: float = 3.14159
decl pi_approx: int = (int) pi
```

4.1 Primitive Types

These types are built-in to the compiler.

- `byte` – represents an unsigned 8-bit integer.
- `int` – represents a signed 32-bit integer.
- `uint` – represents an unsigned 32-bit integer.
- `word` – represents a signed 64-bit integer (a processor word).
- `uword` – represents an unsigned 64-bit integer (a processor word).
- `float` – represents a 32-bit floating-point number.
- `double` – represents a 64-bit floating-point number.

Numeric Literals Numbers are specified as a sequence of digits. A different base may be specified by prefixing the literal with `0`x where x is one of

- `b` – binary, base-2.
- `o` – octal, base-8.
- `d` – decimal, base-10 (the default).
- `x` – hexadecimal, base-16.

By default, integer literals are `int`s, unless the value exceeds the integer capacity, or the literal has a `w` suffix.

Decimal Literals A numeric literal becomes a float if a decimal point '`.`' is encountered.
For floating points, the default is `double` unless `f` is suffixed.

4.2 User-Defined Types

These are types defined using the `data` keyword, with the syntax

```
decl data Name
data Name {
    field1: type1,
    ...
}
```

That is, the datatype `Name` contains the listed fields, which are the listed types. Field declarations are separated by newlines, or by commas.

Member Access Members may then be accessed via the dot '`.`' operator.

```
data Vec { x: int, y: int }
decl v: Vec
v.x // => 0
```

As Parameters Variables of a user-defined datatype are passed around as references, meaning that modifications to said parameters modify the original. For example,

```
func set(v: Vec, n: int) {
    v.x = n
    v.y = n
}

decl v: Vec // v.x = 0
set(v, 5) // v.x = 5
```

Casting Values of user-defined datatypes cannot be cast between eachother. If necessary, cast to a pointer type first.

```
decl data Vec2, Vec3
decl v: Vec2
// ...
decl v: Vec3 = *(*Vec3) @v
```

4.3 Pointers

Pointers are special types which contain the memory address (location) of a variable of some type. Pointers are declared with a star, followed by the datatype at the location. For example,

```
decl n: int = 42
decl p: *int = @n
```

If the type is unknown, one uses the special `*unknown` type. Note that pointers themselves are nothing but integers under the hood, specifically a `uword`.

Creating Pointers The memory location of a variable may be retrieved using the address-of '`@`' unary operator. Note, such operators are evaluated at compile-time. As seen above, the variable t produces pointer $*t$.

Pointer Arithmetic Pointers supports both the addition '`+`' and subtraction '`-`' operators with integers. Such operations are considered to intend "move the pointer left/right by n units", the unit size dependent on the pointer type.

```
decl p: *int = 10
p + 2 // p + 2 * sizeof(int) = 18
// vs
decl p: *byte = 10
p + 2 // p + 2 * sizeof(byte) = 12
```

Pointer Dereferencing This refers to retrieving a value at a pointer, and is done via the star '`*`' unary operator. As expected of the complement of `@`, this strips a star from the type, and hence is only applied to pointer types. Following the example, the value of `n` may be recovered by

```
decl n2: int = *p
```

Casting As pointers are but integers, they may be cast as such. That is, integers may be cast to pointers with any number of stars, and vice versa.

```
decl n: int = 5,
     pint: *int = n,
     pfloat: *float = pint // would also work with '= n'
```

4.4 Arrays

Arrays are contiguous blocks of memory which may hold a sequence of data of one type. Essentially, arrays are pointers, except `sizeof` returns the size in bytes of the array, not of the pointer type.
An array type is declared by suffixing the type with square brackets '`[]`'. Note that the pointer specifier '`*`' is more binding than the array specifier. That is, `*int[]` is a pointer to an array of integers, whereas `(*int)[]` is an array of integer pointers.
The array size is optionally given between the square brackets.

```
decl nums: int[5]
sizeof(nums) // -> 5 * sizeof(int) = 20
```

Note that an arrays size must be known at compile time (e.g, a macro or a constant with known value). If a size is not specified, the declaration **must** be initialised, from which the size will be deduced.

```
decl nums: int[] = { 1, 2, 3, 4, 5 }
sizeof(nums) // -> 5 * sizeof(int) = 20
```

4.5 Constants

If a type is preceded by the `const` keyword, this type is marked as constant and any attempted changes to this type is forbidden. Additionally, attempting to strip a `const` type of its constant status is forbidden (but copying to a non-constant is allowed).

```
decl pi: const float = 3.14159
pi = 5 // error! 'pi' is marked const
decl pi: float = pi
pi = 5 // permitted, as shadow is not const
```

5 Variables

Variables are but labels to reserved location in memory. When defined, variables are assigned a name and a datatype, which dictates the size in bytes of the reserved location. An example would be

```
decl x: int
```

Values may be assigned to variables using the assignment operator '`=`'. Note the type coercion/casting behaviours described previously.

5.1 Multi-Declaration

Commas may be used to separate declarations and, hence, declare multiple symbols per keyword. Each declaration may be of a different type.

```
decl a: byte , b: int , c: word
```

5.2 Scope

"Scope" refers to where a variable exists. The global scope is the top-most scope where all top-level functions and variable reside. Symbols in the global scope may be accessed anywhere in the program.
On the other hand, local scope is not all-encompassing. A new local scope is introduced in every block. Variables defined in such a scope are only accessible from within that function; referencing them outside will result in an error. When a variable is referenced, the scopes will be searched as a stack; that is, local first, global last.

```
decl n: int = 0

func f1 {
    n++ // this will increment the global 'n'
}

func f2 {
    decl n: int = 1
    n++ // this will increment the local 'n' declared above
}
```

To see an example of creating a local scope that is not a function definition:

```
decl n: int = 0 // n = 0

{
    decl n: int = 2 // n = 2
    n++ // n = 3
}

// n = 0
```

6 Functions

Functions are name-associated sections of code which may be called, possible with arguments, and may return a value. They are defined using the `func` keyword. For use before definitions, signatures may be declared using the compound `decl func` keyword.

```
decl func add(int, int) -> int
func add(a: int, b: int) {
    return a + b
}
```

- In declarations, parameter names are not required.
- If no parameters are required, it is possible to omit the brackets entirely.
- If no return type is required, omit the arrow '`->`' and the type.
- If declared, the definition does not require a return type as this can be inferred from its declared signature.

6.1 Overloading

Function overloading is supported, meaning that a function name may be re-used with a different signature. For example,

```
decl func add(int, int) -> int
decl func add(float, float) -> float
```

6.2 Entry Point

All programs have an entry point. By default, it is `main`, taking zero or more integers, and optionally returning an integer.
A new entry point may be defined using the `entry` keyword, by following the keyword by the entry point's name and type. Note, this is a function signature.

```
entry start(int) -> int
```

Only one entry point per program is permitted. Once encountered, future encounters of `entry` will result in an error.

6.3 Compile-Time Functions

These "functions" are resolved in the compilation stage.

`sizeof(`t`)` This returns the size, in bytes, of the argument t. t may be a type name, or a variable, in which case the size of the variable's type will be calculated.

```
sizeof(int) // -> 4
```

`register(`r`)` This may be used in expressions, and returns the contents of register r as a word. r is the name of a register, same as in the assembly code but without the dollar '`$`'.

```
register(sp) // reads $sp
```