Semantics of the C-- Programming Language

CS 473

1 Introduction

This document describes the semantics of the programming language C--. C--is a subset of C, and hence, most C--statements have the same semantics as if they were written in C. The primary difference between the two languages is in which statements are allowed. The syntax specification eliminates a lot of C features, such as float-point numbers, structures, pointers etc. This document specifies which syntactically valid C--programs are also semantically valid (i.e., should compile without errors), and what the effect of executing semantically valid programs is. The topics covered in this document are:

- Summary of Features — what is and isn't a C--program
- Semantic Checking — how to determine if a C--program is semantically sound

Another document will follow which will describe the runtime organization of C--. The C--syntax definition given in the project descriptions 1 and 2, and this document will be the reference for any issues on the C--language.

Please report any bugs, ambiguities, or other problems with this document to me.

2 C--Features

2.1 Supported Features

All C--code is present in one file. (Strictly speaking, the code can be spread over several files, but we do not consider this aspect in the class project.) Also, there are no include files.

The C--language includes the following features:

- **Literals** of type `int`, `void`.
- **Binary integer operators** `+ - * /`
- **Assignment operator** `=`
- **Equality operators** `== !=`
- **Integer relational operators** `< <= >= >`

- **One-dimensional Arrays**
- **Array Element Lookup** using expressions of the form $A[i]$ to fetch the ith element of array A.
- **Functions** which are functions defined in the file.
- **Function invocation** by passing arguments to a function.
- **Expressions** combining all of the above features with precedence rules* and the ability to use parentheses to override precedence.
- **While loops** which repeatedly execute a statement as long as a given boolean expression remains true.
- **If statements** which conditionally execute a statement when a given boolean expression holds true. The if-statement may optionally have an else statement which executed only if the condition holds false.
- **Block statements** which group a sequence of statements together, with optional local variable definitions.
- **Return statements** which terminate execution of a method and return a value back to the spot from which the method was invoked.

3 Declaration Processing

Each declaration of a method, or variable takes a name and associates it with a type. C--programs may then refer to the declared items by giving the name. Your compiler will generate a *symbol* for each declaration. The symbols should be organized into symbol tables, that was done in project 3. The purpose of a symbol table is to make it easy to find the type (as well as other information) associated with a name, paying particular attention to the *scope* of each name. The scope of a name is the region of the program where a particular name is visible.

C--allows expressions and types to contain forward references to functions which are declared later in the source file. Hence, compilation of a C--program is best done in (at least) two passes:

1. Parse the input file, process all declarations and detect duplicate declarations, build abstract syntax trees and symbol tables, and resolve variable names.
2. Resolve all references to functions, check for type mismatches and other semantic errors.
3. Generate code after these two steps.

These three bullet items will be the next three projects in our class.

*The precedence rules are detailed in Syntax of the C--Programming Language

3.1 File Declarations

The top-level of a C--program is a sequence of zero or more (global) variable declarations. After each variable declaration appear one or more field and method declarations. For example, consider:

```
 int x;
int value(void) {
  x = 0;
  return x;
}
int main(void){

   return value();
}
```

Although this program isn't very interesting, it assists in explaining the above statements. This declares the variable `x`. two methods `value` and `main`.

3.1.1 Declaration Visibility

All method declarations are globally visible (and forward references to yet-to-be-defined methods are allowed). No two methods can have the same name. Declaring two methods with the same name is an error.

3.2 Variable Declarations

The declaration of a field gives the type, and then the name of the variable. A variable declaration can occur in the global scope, at the beginning of a method, or at the beginning of a block.

For example, consider:

```
int member;
int offices[20];
```

In this example, a variable `member` whose type is int, and a variable `offices` whose type is array.

Variable declarations are subject to the following restrictions:

- The variable should not have an initializer (a separate initialization statement must be used, and this is taken care of by the grammar).
- Each scope can define only one variable or method with a given name. Obviously, functions are defined in only the global scope, which means that a function and a variable with the same name cannot co-exist in the global scope. Variables defined in "inner" scopes shadow the ones defined in "outer" scopes. The symbol table was structured to address such scoping.

3.3 Local Variable Declarations

Local variables are variables that belong to a particular method. Declarations are similar in syntax to global declarations, Each local variable has a type and name. For example:

```
int fac(int n)
{
  int i;
  int f;

  i = 2;
  f = 1;
  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```

In this example, the function `fac` has three local variables. The first is the formal parameter `n`. The other two are `i` and `f`. The type of all three declarations is `int`.

3.4 Function Definitions

The definition of a function is as follows: First appears a return type (void or int), second the the function, third the formal parameter declarations, and finally the body of the method. For example:

```
  int disp(int x)
  {
     return x+2;
   }
```

Function `disp` can be explained as follows:

- `disp` has one formal parameter which is an integer named `x` and returns an `int`.