Cigrid Language Reference Manual

Version 0.08, Monday 8th November, 2021

David Broman
KTH Royal Institute of Technology

Abstract

This document defines the syntax of the Cigrid programming language. The Cigrid language is a small subset of C/C++ that is aimed for education of compilers. The specification consists of two parts: (i) a concrete syntax definition, and (ii) an abstract syntax definition. The design goal of the language is to keep it small for teaching and learning purposes, as well as expressive enough for constructing interesting programs.

Contents

1 Concrete Syntax

This section describes the concrete syntax of a Cigrid program. Note that a valid Cigrid program can *always* be compiled and executed as a C++ program using for instance the GCC `g++` compiler. Hence, a Cigrid program has the same file ending as a C++ source file, that is, `.cpp`

1.1 Lexical Conventions

A source code file is broken up into a sequence of *tokens*. There are six kinds of tokens: *identifiers*, *keywords*, *constants*, *operators*, and *separators*. Characters between tokens—including *whitespace* and *comments*—are ignored.

Character encoding

The input file is assumed to be encoded using `UTF-8`.Tokens make only use of ASCII characters that are encoded using the least seven significant bits of a byte. Other Unicode characters that are encoded using multiple bytes may appear inside comments, but not in any token.

Whitespace

\t \n

White space characters include tab (ASCII code `0x09`), space (`0x20`), line feed (`0x0A`), and carriage return (`0xD`). No other characters are seen as whitespace.

\r

Comments

There are two kinds of comments: *line comments* and *multi-line comments*. Line comments start with the character sequence `//`. All following characters are ignored, up and until line feed (`0x0A`). A multi-line comment starts with `/*` and ends with `*/`. Multi-line comments are not allowed to be nested.

Identifiers

An *identifier* consists of a sequence of digits, letters, and underscore characters. The first character of an identifier must be a letter or an underscore. Upper-case letters and lower-case letters are distinct. For instance, identifiers `_foo3` and `_Foo3` represent two different identifiers. An identifier can be represented by the following regular expression:
`[_a-zA-Z][_a-zA-Z0-9]*`

Keywords

A *keyword* is a reserved identifier that is used as a terminal symbol during parsing and cannot be used in other contexts. The following keywords are reserved:

```
break     extern    new       while
char      for       return
delete    if        struct
else      int       void
```

A keyword always covers a whole identifier. For instance the sequence `if foo` generates two tokens, where the first token is a keyword and the second token an identifier. By contrast, `iffoo` results in only one identifier token.

Integer Constants

A *decimal integer constant* consists of a sequence of digits. If the sequence consists of more than one character, the first character is not allowed to be zero (character `0`)[1]. The regular expression for a decimal integer is: `0|[1-9][0-9]*` Hexadecimal numbers are proceeded with `0x` or `0X`. The regular expression for hexadecimal numbers is: `0[xX][0-9a-fA-F]+`

Character Constants

A *character constant* represents one character, enclosed within single quotes. For instance, the letter A is written as `'A'`. Single ASCII characters are allowed between `' '` (`0x20`) 32 and `'~'` (`0x7e`) 126, except characters `'\'` (`0x5c`) 92, `'''` (`0x27`) 39, and `'"'` (`0x22`) 34. The following characters cannot be written directly, and must be escaped using the escape character \

newline	`\n`
horizontal tab	`\t`
backslash	`\\`
single quote	`\'`
double quote	`\"`

34 39 92

32 33 35-38 40-91 93-126

'' ! # & ([ ] ~

String Constant

A sequence of characters enclosed by double quotes represents a *string constant*. A string constant uses the same characters and escape characters as character constants. For instance, `"foo"` and `"foo bar\n"` are string constants.

1.2 Precedence Rules

To disambiguate the grammar described in Section 1.3, the following order of precedence and associativity needs to hold. The top of the list has highest precedence. Operators at the same row have the same precedence.

Operators	Associativity	Operator Arity
`!` `~` `-`	Right	Unary
`*` `/` `%`	Left	Binary
`+` `-`	Left	Binary
`<<` `>>`	Left	Binary
`<` `>` `<=` `>=`	Left	Binary
`==` `!=`	Left	Binary
`&`	Left	Binary
`|`	Left	Binary
`&&`	Left	Binary
`||`	Left	Binary

1.3 Syntax

The syntax of Cigrid is described using a variant of Extended Backus-Naur form. Nonterminals are written using italic font (for instance *expr*) and keywords and separation characters within double quotes (for instance `"while"` and `","`). We use the typewriter font for tokens carrying

[1] The reason for this rule is that Cigrid does not support octal numbers. As a consequence, note that a string `00` means that the scanner generates two integer constant tokens.

a value: `Ident` for an identifier, `UInt` for an unsigned integer constant, `Char` for a character constant, and `String` for a string constant. Repetition of zero or more terminal or nonterminal symbols are enclosed by curly braces {...}. An optional symbol is enclosed within square brackets [...].

The grammar for the Cigrid language is as follows:

标红的nonterminal都拥有自己的AST类型

```
      unop → "!" | "~" | "-"                                              (1)
     binop → "+" | "-" | "*" | "/" | "%"                                  (2)
           | "<" | ">" | "<=" | ">=" | "==" | "!="                        (3)
           | "&" | "|" | "&&" | "||"                                      (4)
           | "<<" | ">>"                                                  (5)
                                        struct
        ty → "void" | "int" | "char" | Ident | ty "*"   cigrid只支持以new的形式定义struct类型的
                                                        变量，不支持其他方式定义 (6)
      expr → Ident | UInt | Char | String                                 (7)
             Evar(r)  EInt(r)  EChar(c)  EString(r)
           | expr binop expr  EBinOp(bop,e1,e2)                           (8)
           | unop expr  EUnOp(uop,e)                                      (9)
           | Ident "(" [ expr { "," expr } ] ")" ECall(r,e*)              (10)
           | "new" ty "[" expr "]"  ENew(T,e)--cigrid uses this way to define arrays  (11)
           | Ident "[" expr "]" [ "." Ident ] EArrayAccess(r,e,r?)        (12)
           | "(" expr ")"                                                 (13)
      stmt → varassign ";"  SVarDef(T,r,e)  SVarAssign(r,e)  SArrayAssign(r,e,r?,e)  SExpr(e)  (14)
           | "{" { stmt } "}"  SScope(s*)                                 (15)
           | "if" "(" expr ")" stmt [ "else" stmt ]SIf(e,s,s?)            (16)
           | "while" "(" expr ")" stmtSWhile(e,s)                         (17)
           | "break" ";"  SBreak                                          (18)
           | "return" [ expr ] ";"  SReturn(e?)                           (19)
           | "delete" "[" "]" Ident ";" SDelete(r)                        (20)
           | "for" "(" varassign ";" expr ";" assign ")" stmt             (21)
    lvalue → Ident | Ident "[" expr "]" [ "." Ident ] SVarAssign(r,e)  SArrayAssign(r,e,r?,e)  (22)
    assign → Ident "(" [ expr { "," expr } ] ")"SExpr(ECall(r,e*))        (23)
           | lvalue "=" expr | lvalue "++" | lvalue "--"                  (24)
 varassign → ty Ident "=" expr | assign                                   (25)
                  SVarDef(T,r,e)
    params → [ ty Ident { "," ty Ident } ]                                (26)
    global → ty Ident "(" params ")" "{" { stmt } "}"GFuncDef(T,r,(T,r)*,s*)  (27)
           | "extern" ty Ident "(" params ")" ";"GFuncDecl(T,r,(T,r)*)    (28)
           | ty Ident "=" expr ";"  GVarDef(T,r,e)                        (29)
           | "extern" ty Ident ";"  GVarDecl(T,r)                         (30)
           | "struct" Ident "{" { ty Ident ";" } "}" ";"                  (31)
                                                 GStruct(r,(T,r)*)
   program → { global }                                                   (32)
```

void main(Ident...)
{
if (e)
{SVarAssign()}
else
{}

while e stmt

return e
}

The only expression that is allowed as a separate statement (39) is a function call (23)

2 Abstract Syntax

This section outlines an abstract syntax for a Cigrid compiler. The purpose of including a definition of an abstract syntax is twofold: (i) to enable the definition of the semantics of Cigrid, and (ii) as a pedagogical tool, where the abstract syntax tree (AST) can be pretty printed and tested for correctness. As a consequence, this section includes both a traditional abstract syntax definition, as well as a concrete syntax that can be used when pretty printing an AST.

2.1 Abstract Syntax Definition

The following grammar defines an abstract syntax for a Cigrid program. The main program is given by p. Let c range over characters, r range over text strings, and i over integers. A list of nonterminals is denoted by an overline. For instance, $\overline{e}$ represents a list of expressions. The hat notation $\hat{e}$ denotes an optional term. That is, either expression e exists, or it is marked as not existing. In a functional programming language, this is typically implemented using an option type.

$$uop ::= \texttt{!} \mid \sim \mid \texttt{-} \tag{33}$$

$$bop ::= \texttt{+} \mid \texttt{-} \mid \star \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<<} \mid \texttt{>>} \tag{34}$$

$$T ::= \texttt{TVoid} \mid \texttt{TInt} \mid \texttt{TChar} \mid \texttt{TIdent}(r) \mid \texttt{TPoint}(T) \tag{35}$$

$$e ::= \texttt{EVar}(r) \mid \texttt{EInt}(i) \mid \texttt{EChar}(c) \mid \texttt{EString}(r) \tag{36}$$

$$\mid \texttt{EBinOp}(bop, e, e) \mid \texttt{EUnOp}(uop, e) \mid \texttt{ECall}(r, \overline{e}) \tag{37}$$

$$\mid \texttt{ENew}(T, e) \mid \texttt{EArrayAccess}(r, e, \hat{r}) \tag{38}$$

no SArrayDef???

$$s ::= \texttt{SExpr}(e) \mid \texttt{SVarDef}(T, r, e) \mid \texttt{SVarAssign}(r, e) \tag{39}$$

$$\mid \texttt{SArrayAssign}(r, e, \hat{r}, e) \mid \texttt{SScope}(\overline{s}) \mid \texttt{SIf}(e, s, \hat{s}) \tag{40}$$

$$\mid \texttt{SWhile}(e, s) \mid \texttt{SBreak} \mid \texttt{SReturn}(\hat{e}) \mid \texttt{SDelete}(r) \tag{41}$$

e

$$g ::= \texttt{GFuncDef}(T, r, \overline{(T, r)}, s) \mid \texttt{GFuncDecl}(T, r, \overline{(T, r)}) \tag{42}$$

这个s要求必须是scope

$$\mid \texttt{GVarDef}(T, r, e) \mid \texttt{GVarDecl}(T, r) \mid \texttt{GStruct}(r, \overline{(T, r)}) \tag{43}$$

$$p ::= \texttt{Prog}(\overline{g}) \tag{44}$$

e

2.2 Relation to the Concrete Syntax

The following items highlight some key connections between the concrete syntax defined in Section 1.3 and the abstract syntax in Section 2.1. The numbers refer to the different lines in the grammar definitions.

Expressions and Types

- There is a direct match between unary operators (1) and (33), binary operators (2-5) and (34), as well as between types (6) and (35).

- The first four constructors of expressions are also direct: (7) corresponds to (36).

- Binary operators (8) are written in infix form in the concrete syntax, but in prefix form in the abstract syntax (37).

- Function calls (10), for instance `foo(7,8)`, are defined using a constructor $\texttt{ECall}(r, \overline{e})$ (37), where the string represents the called function name, and where the arguments are given as a list of expressions $\overline{e}$.

- The `new` construct (11) creates a new array of elements of a specific type T (38). For instance `new int[10]` creates a new array with 10 integer elements.

- Elements in an array can be accessed using bracket syntax (12). For instance, `bar[8].x` accesses an element with index 8 in an array `bar`, where each element is a `struct` that includes the field label `x`. In the abstract syntax (38) `EArrayAccess`$(r, e, \hat{r})$ represents such array access, where the first element r is the name of the accessed array (`bar` in the example) and e is the expression representing the index. The optional string $\hat{r}$ is the label name if the array access is accessing an array of `struct` elements. If the array access is of another type, for instance an array of integers, then this option value is empty.

Statements

- Variables are always defined with a given value. For instance `int x = 5;`. That is, it is not allowed to have uninitialized variables. Hence, (14) and (25) correspond to `SVarDef`(T, r, e) in (39). Similarly, a variable assignment (24) corresponds to the constructor `SVarAssign`(r, e) in (39).

- The only expression that is allowed as a separate statement (39) is a function call (23).

- Assignments (24) can be done to an *lvalue* (22). For simple variable assignments, such as `x = 7;`, the *lvalue* is an identifier representing a memory location. Assignments of values to arrays have a special syntax (22). For instance, the statement `bar[3].x = 5;` assigns a value 5 to a field with label `x` in an array named `bar`. The corresponding abstract syntax is `SArrayAssign`$(r, e, \hat{r}, e)$. The first element r is the name of the array and the second element e is the index in the array. The last element e is the expression that is evaluated to a value before it is assigned. The third element $\hat{r}$ is an optional name of the label, if the array element type is a `struct`.

- A scope (15) makes it possible to create a sequence of statements. Scopes also create local variables, which are not live outside the scope. The corresponding abstract syntax constructor `SScope`$(\overline{s})$ includes a list of statements $\overline{s}$.

- An `if` statement (16) is represented in the abstract syntax by `SIf`$(e, s, \hat{s})$. Note how the else branch $\hat{s}$ is optional.

- Syntax for `while` loops (17), `break` statements (18), `return` statements (19), and `delete` statements (20) directly translate to the corresponding abstract syntax constructors (41).

- The `for` statement (21) does not have a corresponding abstract syntax constructor. Instead, a `for` statement can be translated into a `while` statement, including scopes. Likewise, the increment `++` and decrement `--` operators (24) can be translated into a combination of assignment and expression constructs.

Globals

- Functions are defined with a function body (27), with the corresponding abstract syntax constructor `GFuncDef`$(T, r, \overline{(T, r)}, s)$. The parameters are defined using a list of tuples $\overline{(T, r)}$. Also, note that the concrete syntax (27) expects a list of statements $\{$ *stmt* $\}$, whereas the abstract syntax expects one statement s. The reason for this difference is that a function body needs to enforce the use of curly brackets. Hence, after parsing a list of statements, encode it as a scope statement in the abstract syntax.

- Functions can also be declared without defining a body. Declarations are always external in Cigrid (28) meaning that the names are available outside the compilation unit. The abstract syntax is given in (42).

- Global variables can either be defined (29) or declared (30). Defined variables always contain an initial value. The corresponding abstract syntax definitions are found in (43).

- New structures can only be declared (31) and cannot be given default values. The corresponding abstract syntax constructor is $\texttt{GStruct}(r, \overline{(T, r)})$, shown in (43). Note how the record fields are declared using a list of tuples $\overline{(T, r)}$.

- A Cigrid program consists of a sequence of global definitions and declarations (32), represented using the abstract syntax construct $\texttt{Prog}(\overline{g})$ (44).

2.3 Pretty Printing

To be able to pretty print the abstract syntax definition in Section 2.1 unambiguously, a grammar of the concrete syntax of the abstract syntax is needed. This section contains such grammar. The translation between this grammar, and the grammar given in Section 2.1 is direct. Only a few remarks are needed:

- The terminal symbol `TextStr` is used when a string r in the AST is printed. The printed output must be properly escaped (according to Section 1.1) and enclosed by `"` characters. For instance, if a string contains the keyword `Sigrid`, the pretty printer should output `"Sigrid"`.

- The terminal symbol `Integer` is used when the AST contains an integer. The integer is signed and can contain negative numbers. The output should be in decimal form. Examples of outputs are `0`, `-1231` and `889`.

- The terminal symbol `TextChar` is used for characters. The printed character must be properly escaped and enclosed by `'` characters. For instance, a character `A` is printed as `'A'`, and a new line (line feed) as `'\n'`.

The actual concrete syntax for pretty printing is as follows:

$$
\begin{align}
uop &\rightarrow \texttt{"!"} \mid \texttt{"\textasciitilde"} \mid \texttt{"-"} \tag{45}\\
bop &\rightarrow \texttt{"+"} \mid \texttt{"-"} \mid \texttt{"*"} \mid \texttt{"/"} \mid \texttt{"\%"} \tag{46}\\
&\mid \texttt{"<"} \mid \texttt{">"} \mid \texttt{"<="} \mid \texttt{">="} \mid \texttt{"=="} \mid \texttt{"!="} \tag{47}\\
&\mid \texttt{"\&"} \mid \texttt{"|"} \mid \texttt{"\&\&"} \mid \texttt{"||"} \tag{48}\\
&\mid \texttt{"<<"} \mid \texttt{">>"} \tag{49}\\
T &\rightarrow \texttt{"TVoid"} \mid \texttt{"TInt"} \mid \texttt{"TChar"} \tag{50}\\
&\mid \texttt{"TIdent"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{")"} \tag{51}\\
&\mid \texttt{"TPoint"}\ \texttt{"("}\ T\ \texttt{")"} \tag{52}\\
e &\rightarrow \texttt{"EVar"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{")"} \tag{53}\\
&\mid \texttt{"EInt"}\ \texttt{"("}\ \texttt{Integer}\ \texttt{")"} \tag{54}\\
&\mid \texttt{"EChar"}\ \texttt{"("}\ \texttt{TextChar}\ \texttt{")"} \tag{55}\\
&\mid \texttt{"EString"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{")"} \tag{56}\\
&\mid \texttt{"EBinOp"}\ \texttt{"("}\ bop\ \texttt{","}\ e\ \texttt{","}\ e\ \texttt{")"} \tag{57}\\
&\mid \texttt{"EUnOp"}\ \texttt{"("}\ uop\ \texttt{","}\ e\ \texttt{")"} \tag{58}\\
&\mid \texttt{"ECall"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{","}\ \texttt{"\{"}\ \{\ e\ \}\ \texttt{"\}"}\ \texttt{")"} \tag{59}\\
&\mid \texttt{"ENew"}\ \texttt{"("}\ T\ \texttt{","}\ e\ \texttt{")"} \tag{60}\\
&\mid \texttt{"EArrayAccess"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{","}\ e\ \texttt{","}\ [\,\texttt{TextStr}\,]\ \texttt{")"} \tag{61}\\
s &\rightarrow \texttt{"SExpr"}\ \texttt{"("}\ e\ \texttt{")"} \tag{62}\\
&\mid \texttt{"SVarDef"}\ \texttt{"("}\ T\ \texttt{","}\ \texttt{TextStr}\ \texttt{","}\ e\ \texttt{")"} \tag{63}\\
&\mid \texttt{"SVarAssign"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{","}\ e\ \texttt{")"} \tag{64}\\
&\mid \texttt{"SArrayAssign"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{","}\ e\ \texttt{","}\ [\,\texttt{TextStr}\,]\ \texttt{","}\ e\ \texttt{")"} \tag{65}\\
&\mid \texttt{"SScope"}\ \texttt{"("}\ \texttt{"\{"}\ \{\ s\ \}\ \texttt{"\}"}\ \texttt{")"} \tag{66}\\
&\mid \texttt{"SIf"}\ \texttt{"("}\ e\ \texttt{","}\ s\ \texttt{","}\ [\,s\,]\ \texttt{")"} \tag{67}\\
&\mid \texttt{"SWhile"}\ \texttt{"("}\ e\ \texttt{","}\ s\ \texttt{")"} \tag{68}\\
&\mid \texttt{"SBreak"} \tag{69}\\
&\mid \texttt{"SReturn"}\ \texttt{"("}\ [\,e\,]\ \texttt{")"} \tag{70}\\
&\mid \texttt{"SDelete"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{")"} \tag{71}\\
l &\rightarrow \texttt{"\{"}\ \{\texttt{"("}ty\ \texttt{","}\ \texttt{TextStr}\ \texttt{")"}\}\ \texttt{"\}"} \tag{72}\\
g &\rightarrow \texttt{"GFuncDef"}\ \texttt{"("}\ T\ \texttt{","}\ \texttt{TextStr}\ \texttt{","}\ l\ \texttt{","}\ s\texttt{")"} \tag{73}\\
&\mid \texttt{"GFuncDecl"}\ \texttt{"("}\ T\ \texttt{","}\ \texttt{TextStr}\ \texttt{","}\ l\ \texttt{")"} \tag{74}\\
&\mid \texttt{"GVarDef"}\ \texttt{"("}\ T\ \texttt{","}\ \texttt{TextStr}\ \texttt{","}\ e\ \texttt{")"} \tag{75}\\
&\mid \texttt{"GVarDecl"}\ \texttt{"("}\ T\ \texttt{","}\ \texttt{TextStr}\ \texttt{")"} \tag{76}\\
&\mid \texttt{"GStruct"}\ \texttt{"("}\ \texttt{TextStr}\ \texttt{","}\ l\ \texttt{")"} \tag{77}\\
p &\rightarrow \{g\} \tag{78}
\end{align}
$$

3 Semantics

The following section informally describes the semantics of Cigrid.

3.1 Dynamic Semantics

The dynamic semantics (how a program behaves when running the compiled program) is indirectly specified by the C++ standard. To examine the runtime behavior, please create a bash script `cigrid-gcc` with the content:

```
#!/bin/bash
g++ "$@" -Wno-dangling-else \
-Wno-c++11-compat-deprecated-writable-strings
```

If a program passes the checks of the static semantics (see below) and it can be compiled with the above script, then its runtime behavior corresponds to Cigrid's dynamic semantics.

3.2 Static Semantics

The static semantics of Cigrid rules out programs that do not follow certain name analysis checks, as well as type checks. We do not give a formal semantics (such as formal type rules) here. Instead, the following informal rules define the static semantics of Cigrid. That is, the rules state what must hold. If any of these rules do not hold, then a static error must be reported by the compiler.

Requirements on names:

1. All identifiers defined in the global scope of a file must be unique. That is, no pairs of defined function names, defined global variable names, or `struct` names are allowed to be identical within a file.

2. If a function and a global variable are both declared and defined, they need to have the same signature. It is not allowed to declare or define a function or a global variable more than once.

3. All parameter names in a function definition or function declaration must have unique names (within the same function).

4. All variable names need to be *declared* before use. This includes locally declared variables within functions, parameter names, and global variables. Note that a function or a global variable is allowed be *defined* after its use, as long as it is *declared* before use. See the AST nodes, e.g., `GFuncDecl` (function declaration) vs. `GFuncDef` (function definition, which also includes the body).

Requirements on types:

1. Only `==` and `!=` operations are allowed on pointer types. All other operations on pointers are not allowed.

2. In a pointer, where the pointer type is not a built-in type, the name has to be declared as a `struct`. For instance, in `Tree* node(Tree* left, int x, Tree* right)` the abstract syntax for the pointer type is `TPoint(TIdent("Tree"))`. In that case, `Tree` has to have been declared earlier in the file as a `struct`.

3. The field name must be a valid field in the struct in struct projection. For instance, in `foo[0].bar` the name `foo` must be of type `TPoint(TIdent(id))` where `id` has a field called `bar`.

4. Array indexing is only allowed on pointer types. For instance, if we have `foo[x]` then `foo` must be of type `TPoint(T)`, for some type `T`.

5. The type of the return value in a `return` statement needs to be equivalent to the return type defined in the function signature. If `void` is given in the function signature, the function should either have no `return` statement or a `return` statement without a value.

Note that you do not have to check for type collisions between `char` and `int` types since Cigrid allows implicit conversions (as in C++).

Acknowledgements

I would like to thank the teaching assistants of the compiler course ID2202 at KTH for their comments and suggested corrections: Linnea Ingmar, Viktor Palmkvist, and Lars Hummelgren.