≡ Listen ▶

THE UNIVERSITY OF ARIZONA
DEPARTMENT OF COMPUTER SCIENCE

CSc 453: C-- Language Specification

Lexical rules | Syntax rules | Semantic checking | Execution behavior

Notation: The lexical and syntax rules given below use a notation called EBNF to specify patterns. EBNF notation characters are written in magenta. You can find more information about EBNF here.

1. Lexical Rules

1.1. Notation

To reduce clutter, we use the following names in the patterns listed below:

letter ::= `a | b | ... | z | A | B | ... | Z`
digit ::= `0 | 1 | ... | 9`

1.2. Tokens

The tokens of the language are as follows:

Name	Pattern	Comments
ID	*letter* { *letter* | *digit* | _ }	identifier: e.g., **x**, **abc**, **p_q_12**
INTCON	*digit* { *digit* }	integer constant: e.g., **12345**
LPAREN	**(**	Left parenthesis
RPAREN	**)**	Right parenthesis
LBRACE	**{**	Left curly brace
RBRACE	**}**	Right curly brace
COMMA	**,**	Comma
SEMI	**;**	Semicolon
kwINT	**int**	Keyword: **int**
kwIF	**if**	Keyword: **if**
kwELSE	**else**	Keyword: **else**
kwWHILE	**while**	Keyword: **while**
kwRETURN	**return**	Keyword: **return**
opASSG	**=**	Op: Assignment
opADD	**+**	Op: addition
opSUB	**–**	Op: subtraction
opMUL	*****	Op: multiplication
opDIV	**/**	Op: division
opEQ	**==**	Op: equals
opNE	**!=**	Op: not-equals
opGT	**>**	Op: greater-than
opGE	**>=**	Op: greater than or equal
opLT	**<**	Op: less-than

opLE	**<=**	Op: less than or equal
opAND	**&&**	Op: logical and
opOR	**||**	Op: logical or

1.3. Comments

Comments are as in C, i.e. a sequence of characters preceded by `/*` and followed by `*/`, and not containing any occurrence of `*/`.

[Back to top]

2. Syntax Rules

Nonterminals are shown in lower-case italics; terminals are shown in boldface or upper-case. The symbol ‘ε’ (“epsilon”) denotes the empty sequence.

The grammar rules for full C-- are given below. Changes to the grammer relative to that for the G2 subset of the language are shown highlighted here.

The start symbol of the grammar is *prog*.

2.1 Grammar Productions

prog	: *func_defn* *prog*
	| *var_decl* *prog*
	| ε
var_decl	: *type* *id_list* SEMI
id_list	: ID
	| ID COMMA *id_list*
type	: kwINT
func_defn	: *type* ID LPAREN *opt_formals* RPAREN LBRACE *opt_var_decls* *opt_stmt_list* RBRACE
opt_formals	: ε
	| *formals*
formals	: *type* ID COMMA *formals*
	| *type* ID
opt_var_decls	: ε
	| *var_decl* *opt_var_decls*
opt_stmt_list	: *stmt* *opt_stmt_list*
	| ε
stmt	: *fn_call* SEMI
	| *while_stmt*
	| *if_stmt*
	| *assg_stmt*
	| *return_stmt*
	| LBRACE *opt_stmt_list* RBRACE
	| SEMI

if_stmt	: kwIF LPAREN *bool_exp* RPAREN *stmt*
	| kwIF LPAREN *bool_exp* RPAREN *stmt* kwELSE *stmt*
while_stmt	: kwWHILE LPAREN *bool_exp* RPAREN *stmt*
return_stmt	: kwRETURN SEMI
	: kwRETURN *arith_exp* SEMI
assg_stmt	: ID opASSG *arith_exp* SEMI
fn_call	: ID LPAREN *opt_expr_list* RPAREN
opt_expr_list	: ε
	| *expr_list*
expr_list	: *arith_exp* COMMA *expr_list*
	| *arith_exp*
arith_exp	: ID
	| INTCON
	| *arith_exp* *arithop* *arith_exp*
	| LPAREN *arith_exp* RPAREN
	| opSUB *arith_exp*
	| *fn_call*
bool_exp	: *arith_exp* *relop* *arith_exp*
	| *bool_exp* *logical_op* *bool_exp*
arithop	: opADD
	| opSUB
	| opMUL
	| opDIV
relop	: opEQ
	| opNE
	| opLE
	| opLT
	| opGE
	| opGT
logical_op	: opAND
	| opOR

2.2. Operator Associativities and Precedences

The following table gives the associativities of various operators and their relative precedences. An operator with a higher precedence binds "tighter" than one with lower precedence. Precedences decrease as we go down the table.

2.2.1. Arithmetic Operators

Operator	Associativity
opSUB (unary)	right to left
opMUL, opDIV	left to right
opADD, opSUB (binary)	left to right

2.2.2. Boolean Operators

Operator	**Associativity**
opAND	left to right
opOR	left to right

---

3. Semantic Rules

3.1. Scopes and types

There are two kinds of scope in the C-- subset of C--: (1) *global* scope; and (2) for each function in the program, the scope *local* to that function.

An identifier in C-- has one of two possible types in C--: (1) an **`int`** variable; or (2) a function.

Variables can be declared as globals or as locals. However, the grammar rules of C-- are such that functions can only be defined as globals. Thus, we can have the following possible combinations of scope and type:

	Global	**Local**
Variable	Yes	Yes
Function	Yes	No

3.2. Declarations

The scope of an identifier in a program is given as follows:

1. A variable declared outside a function definition has global scope.
2. A variable declared within a function definition has scope local to that function.
3. The formal parameters of a function have scope local to that function.

The type of an identifier in a program is given as follows:

1. An identifier declared as a function name has type *function*.
2. An identifier declared as a variable (i.e., not as a function) has type *variable*.

3.3. Uses

C-- follows the commonly used rule that a use of an identifier refers to the most deeply nested declaration enclosing that use. Since C-- has only global and local scopes, this translates to the following:

1. If an identifier *x* is declared as a local within a function *f* then uses of *x* within *f* refer to this local.
2. Otherwise, if *x* is declared as a global prior to the definition of the function *f* then uses of *x* within *f* refer to this global.
3. Otherwise, any use of *x* within *f* has no declaration to refer to.

A C-- program must satisfy the following requirements:

1. An identifier can be declared at most once as a global and at most once as a local within any particular function.

   Note that an identifier can be declared as local in multiple functions (since each function has its own distinct local scope).

2. An identifier x that is used within a function body (i.e., which occurs in a statement or expression in the function body) must have been declared prior to the use. The corresponding declaration (see Section 3.3 above) of x must satisfy the following requirements:
   1. If the use of x is as a function call:
      - the corresponding declaration of x should be that of a function; and
      - the number of arguments in the call must be equal to the number of arguments in x's declaration
   2. If the use of x is not a function call, the corresponding declaration of x should be that of a variable.

---

4. Execution behavior

The C-- language has the execution characteristics expected of a C-like block-structured language. The description below mentions only a few specific points that are likely to be of interest. For points not mentioned explicitly, you should consider the behavior of C-- to be as for C.

4.1. Data

An object of type `int` occupies 32 bits.

4.2. Order of Evaluation

1. The evaluation order of the operands of expressions with the following kinds of operators is unspecified: arithmetic operators (`opADD`, `opSUB`, `opMUL`, or `opDIV`) and relational operators (`opEQ`, `opNE`, opGT, `opGE`, `opLT`, `opLE`). This means that the operands of such expressions can be evaluated in any order.

2. Expressions involving the logical operators `opAND` and `opOR` must be evaluated using short circuit evaluation.

3. The order in which the actual parameters in a function call are evaluated is unspecified.

4.3. Program execution

1. Program execution begins at a function named `main()`.

   Note: You are not required to check whether the program being compiled defines a function named `main()`. If a program does not define `main()`, SPIM will generate an error message.

2. Execution returns from a function if either an explicit **return** statement is executed, or if execution "falls off" the end of the function body. In the latter case, no value is returned.

3. Programs will use a function `println()` to print out integer values. Code for this function will not be part of the input program, but will be generated by your compiler as a hard-coded sequence of MIPS instructions as discussed in the document "*Translating Three-Address Code to MIPS Assembly Code*". This function will behave as though it was defined as

   ```
   void println(int x) { printf("%d\n", x); }
   ```

[Back to top]