Listen

THE UNIVERSITY OF ARIZONA.
DEPARTMENT OF COMPUTER SCIENCE

CSc 453: G2 subset of C--

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

The lexical rules and token specifications are as for the G0 subset, as described here.

[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 G2 are given below. Changes to the grammer relative to that for the G1 subset of the language, i.e., grammar symbols and rules that have been added, are shown highlighted.

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

bool_exp        : arith_exp relop arith_exp

arith_exp       : ID
                | INTCON

relop           : opEQ
                | opNE
                | opLE
                | opLT
                | opGE
                | opGT
```

---

3. Semantic Rules

3.1. Scopes and types

There are two kinds of scope in the G2 subset of C--: (1) *global* scope; and (2) for each function in the program, the scope *local* to that function.

An identifier in G2 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 G2 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

G2 follows the commonly used rule that a use of an identifier refers to the most deeply nested declaration enclosing that use. Since G2 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.

3.4. Semantic checking requirements

A G2 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. Program execution

- 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.

- 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]