Compila 19

Language specification

INF5110, spring 2019
IFI

Table of Contents

1 Introduction

This document specifies and describes the syntax and the static semantics of the language *Compila 19*. The dynamic semantics, i.e., the description of the language's behavior when being executed, should be fairly clear even without explicit formal specification.

1.1 Notational conventions and syntax of this document

In the description of the grammar later, we use capital letters for non-terminals. As *meta-symbols* for the grammar, we use the following:

```
->,  |,  (,  ), {,  },  [,  ], "
```

Commas in the previous line are used as "meta-meta symbols" in the enumeration of the meta-symbols.

When writing down the grammar in some variant of EBNF,, `{...}` represents iteration of zero or more times, `[...]` represents *optional* clauses. Everything else, written as contiguous sequences, are *terminal symbols*. Those with only *small* letters are reserved *keywords* of the meta-language.

Note that terminal symbols of the Compila-language are written in "string-quotes" (i.e., with a `"` at the beginning and at the end) to distinguish them from symbols from the meta-language. Some *specific* terminal symbols are written in capitals, and *without* quotes. Those are

- `NAME`,
- `INT_LITERAL`,
- `FLOAT_LITERAL` and
- `STRING_LITERAL`.

See the following section about lexical aspects for what those terminal symbols represent.

2 Lexical aspects

2.1 Identifiers and literals

- `NAME` must start with a letter, followed by a (possibly empty) sequence of numeric characters, letters, and underscore characters; the underscore is not allowed to occur at the end. Capital and small letters are considered different.
- All *keywords* of the languages are written in with lower-case letters. Keyword *cannot* be used for standard identifiers.
- `INT_LITERAL` contains one or more numeric characters.
- `FLOAT_LITERAL` contains one or more numeric characters, followed by a decimal point sign, which is followed by one or more numeric characters.
- `STRING_LITERAL` consists of a string of characters, enclosed in quotation marks (`"`). The string is not allowed to contain line shift, new-line, carriage return, or similar. The semantic *value* of a `STRING_LITERAL` is only the string itself, the quotation marks are not part of the string value itself.

2.2 Comments

Compila supports *single line* and *multi-line* comments.

1. Single-line comments start with `//` and the comment extends until the end of that line (as in, for instance, Java, C++, and most modern C-dialects).
2. Multi-line comments start with `/*` and end with `*/`.

The latter form cannot be nested. The first one is allowed to be "nested" (in the sense that a commented out line can contain another `//` or one of the multi-line comment delimiters, which are then ignored).

3 Data types

3.1 Built in data types

The language has four built-in types and user-defined types:

- *Built-in types*
  1. floating point numbers (`"float"`),
  2. integers (`"int"`),
  3. strings (`"string"`), and
  4. booleans (`"bool"`).
- *User-defined* types:
  1. Each (name of a) record represents a type.
  2. Reference types, representing references to elements of the specified types. The reference type constructor can be nested.

3.2 Records

The language supports records. For people coming from Java/C++ etc., records can be seen as (very) simple form of classes, containing only instance variables as members, but support neither *methods* nor *inheritance* nor explicitly programmable *constructors*. "Instance variables" are more commonly called record fields or just *fields* when dealing with records. Records do support *instantiation*, here via the `new` keyword. Another aspect which resembles classes as in Java is that variables of record type contains either a pointer to an element ("object") of that record type or the special pointer value `null`.[1]

3.3 References

The language allows that variables can be declared to be of reference type ("pointers", if you will) by writing, e.g.,

```
var x: ref(int);
```

Variables, like `x`, can be assigned values that are references, so, e.g., the following is allowed:

```
var y: int;
y := 42;
x := ref(y);
```

Correspondingly, one can "follow" a reference `r` by using `deref(r)`, so that, given the previous definitions, the following is legal:

```
y := deref(x);
```

Expressions with `deref` can also be used as L-values, so that we can assign values to the location that they are pointing to, e.g.:

```
deref(x) := y;
```

See also later the swap procedure example later in the context of parameter passing.

[1] Records are sometimes also called "structs".

4 Syntax

4.1 Grammar

The following productions in EBNF describe the syntax of the language. For precedences and associativity of various constructs, see later.

```
PROGRAM          -> "program" NAME "begin" [ DECL {";" DECL}]  "end"

DECL             -> VAR_DECL | PROC_DECL | REC_DECL

VAR_DECL         -> "var" NAME ":" TYPE

PROC_DECL        -> "proc" NAME
                    "(" [ PARAM_DECL { "," PARAM_DECL } ] ")"
                    [ ":" TYPE ]
                    "begin" [DECL{";" DECL}] [STMT {";" STMT}] "end"

REC_DECL         -> "struct" NAME "{" [ VAR_DECL {";" VAR_DECL }]  "}"

PARAM_DECL       -> NAME ":" TYPE

EXP              -> EXP LOG_OP EXP
                  | "not" EXP
                  | EXP REL_OP EXP
                  | EXP ARIT_OP EXP
                  | LITERAL
                  | CALL_STMT
                  | "new" NAME
                  | VAR
                  | REF_VAR
                  | DEREF_VAR
                  | "(" EXP ")"


REF_VAR          -> "ref" "(" VAR ")"
DEREF_VAR        -> "deref" "(" VAR ")" | "deref" "(" DEREF_VAR ")"

VAR              -> NAME | EXP "." NAME

LOG_OP           -> "&&" | "||"

REL_OP           -> "<" | "<=" | ">" | ">=" | "=" | "<>"

ARIT_OP          -> "+" | "-" | "*" | "/" | "^"

LITERAL          -> FLOAT_LITERAL | INT_LITERAL | STRING_LITERAL
```

```
                    | "true" | "false" | "null"

STMT                -> ASSIGN_STMT
                     | IF_STMT
                     | WHILE_STMT
                     | RETURN_STMT
                     | CALL_STMT

ASSIGN_STMT         -> VAR ":=" EXP | DEREF_VAR ":=" EXP

IF_STMT             -> "if" EXP "then" { STMT ";" } "fi"
                       [ "else"  { STMT ";" } "fi" ]
WHILE_STMT          -> "while" EXP "do"  { STMT ";" } "do"

RETURN_STMT         -> "return" [ EXP ]

CALL_STMT           -> NAME "(" [ EXP { "," EXP } ] ")"

TYPE                -> "float" | "int" | "string" | "bool" | NAME
                     | "ref" "(" TYPE ")"
```

4.2 Precedence

The precedence of the following constructs is ordered as follows (from lowest precedence to the highest):

1. `||`
2. `&&`
3. `not`
4. All relational symbols
5. `+` and `-`
6. `*` and `/`
7. `^` (exponentiation)
8. `.` ("dot", to access fields of a record ("object"))

4.3 Associativity

- The binary operations `||`, `&&`, `+`, `-`, `*`, and `.` are *left-associative*, but exponentiation `^` is right-associative.
- Relation symbols are non-associative. That means that for instance `a < b + c < d` is illegal.
- It's legal to write `not not not b` and it stands for `(not (not (not b)))`.

5 Parameter passing

When describing the parameter passing mechanisms of the language, this document distinguishes (as is commonly done) between

- *actual* parameters and
- *formal* parameters.

The actual parameters are the expressions (which include among other things variables) as part of a procedure *call*. The formal parameters are the variables mentioned as part of procedure *definition*. The language supports supports as only parameter passing mechanism call-by-value:

5.1 Call-by-value

The formal parameters are *local* variables in the procedure definition. When a procedure is being called, the *values* of the local parameters are *copied* into the corresponding formal parameters.[1]

5.2 Small example for call-by-reference

```
program swapexample
  begin
    proc swap (a :  ref(int), b : ref(int))
    begin
      var tmp : int;
      tmp      := deref(a);
      deref(a) := deref(b);
      deref(b) := tmp
    end;
    proc main ( )
      begin
        var x : int;
        var y : int;
        var xr : ref (int);
        var yr : ref (int);
        x := 42; y := 84;
        xr := ref (x); yr := ref(y);
        swap (xr,yr)
      end
  end
```

[1] The language supports reference types. Variables or expressions of reference type are passed *by value*. That leads to a behavior resembling pretty much "call-by-reference", but the parameter passing mechanism proper *is* call-by-value (of reference data). One may speak of call-by-value-reference.

6 Standard library

The programming language comes with a standard library which offers a number of IO-procedures. All reading, i.e., input, is done from standard input ("stdin"). All writing, i.e., output, is to standard output ("stdout")

`proc readint(): int`	read one integer
`proc readfloat(): float`	read one float
`proc readchar(): int`	read one character and return its ASCII value. Return `-1` for `EOF`
`proc readstring(): string`	read a string (until first whitespace
`proc readline(): string`	read one line
`proc printint(i:int)`	write an integer
`proc printfloat(f:float)`	write one floating point number
`proc printstr(s:string)`	write one string
`proc printline(s:string)`	write one string, followed by a "newline"

7 Static semantics / typing / evaluation.

This part is not needed for Oblig 1.

7.1 Binding of names

The using occurence of an identifier (without a preceding dot) is bound in the common way to a *declaration*. This association of the use of an identifier to a declaration ("binding") can be described informally as follows: Look through the block or scope which encloses the use-occurence of the identifer (where the block refers to the procedure body or program). The binding occurrence corresponding to the use occurence is the *first* declaration found in this way. If no binding occurence is found that way, the program is *erroneous*. Formal parameters count as declarations local to the procedure body.

Use occurences of a name preceded by a dot correspond to the clause `EXP "." NAME` in the production for the non-terminal `VAR` in the grammar. Those names are bound by looking at the type of `EXP` (which is required to be a record-type) and look up the field with name `NAME` in that record. It's an error, if `EXP` is not of record-type or else, there is not such field in that record,

7.2 Typing of compound constructs

- **expressions:** expressions need to be checked for type correctness in the obvious manner. The whole expression (if it type-checks) will thus carry a type.
- **assignments:** Both sides of an assignment must be of the same type. Note: it is allowed to assign to the formal parameters of a procedure. That applies to both call-by-value and call-by-reference parameters. Of course, the effect of an assignment in these two mechanisms is different.
- **conditionals and while loop:** the condition (i.e., expression) in the conditional construct must be of type `bool`. Same for the condition in the while loop.
- **field selection:**
  - the expression standing in front of a dot must be of record type.
  - the name standing after a dot are the name of a field/attribute of the record type of the expression in front of the dot. The type of the field selection expression (if it type checks) is the type as declared for the field of the record.

7.3 Types and implicit type conversion

It is allowed to assign an expression of type `int` to a variable of type `float`. The inverse situation is not allowed. There is no type cast operator. If an *arithmetic* expression has at least one operand of type `float`, the operation is evaluated using floating point arithmetic and the result is of type `float`. Exponentiation is *always* considered done with floating point arithmetic and the result is of type `float`.

7.4 Short-circuit evaluation

The logical operators `&&` and `||` use so-called *short-circuit evaluation*. That means that *if* the value of the logical expression can be determined after one has evaluated the *first part, only*, the rest of the expression is *not* evaluated.

8 Procedures

- In a procedure, all declarations are required to occur *before* executable code (statements). In a procedure, the *same* declarations are allowed as on the outermost, global scope, i.e., procedure-local declarations of variables, procedures, and records are allowed.
- Procedures called within an expression *must* have a defined return type. That type must match with the way the call is *used* in an expression.
- Concerning the number and types of the parameters of a procedure: they must coincide comparing the declaration/definition of the procedure and the use of a procedure. That requirement applies also to the parameter passing mechanism (i.e., whether the variable resp. actual parameter is marked as "by `ref`".
- Return statements:
  - A `return`-statment is allowed only in procedure-definitions. Such a statement marks that the procedure terminates (and returns). In addition, the return statement gives an expression for the value to be returned to the caller.
  - If a procedure is declared without return type, the procedure does not *need* a return statement. In that case, the procedure returns (without a return value) when the last statement in the procedure body has been executed.
  - If a procedure has declared a return type, its body is required to have a return statement (with corresponding expression of the correct type). That statement need to be the last statement in the procedure's body.

9 Further conditions

- Declarations must be *unique* per block. Two declarations (within one block) of a procedure, a record, or a variable with the same name are considered as double declarations, which are *forbidden*.
- The name of a formal parameter must not collide with names of local declarations within the procedure. Besides, the names of all formal parameters of one procedure must by distinct.
- All names being used must be declared.
- Each program must have a procedure named `main`. This procedure is the one called upon start of the program.