Compila16

Language specification

INF5110, spring 2016
IFI

Table of Contents

1 Introduction

This document specifies and describes the syntax and the static semantics of the language *Compila16*. The dynamic semantics, i.e., the description of the language's behavior when being executed, should be fairly clear even without explicit specification, at least, clear enough for the first oblig which concerns itself with the front-end of the compiler, in particular, the syntactic aspects of the language, i.e., the lexer and the parser. Further details concerning the dynamic semantics might follow in connection with the second oblig.

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 (commas used as "meta-meta symbols" in the enumeration):

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

Here, `{...}` represents iteration of zero or more times, `[...]` representions *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" (with a `"` at the beginning and 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 exactly 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. Capital and small letters are considered different.
- All *keywords* of the languages are written in with lower-case letters. They *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

Comments start with `//` and the comment extends until the end of that line (as in, for instance, Java, C`++`, and most modern C-dialects)

3 Data types

3.1 Built in data types

The language has 4 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: Each (name of a) class represents a type

3.2 Classes

The language supports a (very) simple form of classes. The classes only contain instance variables as members, but neither supports *methods* nor *inheritance* nor explicitly programmable *constructors*. They support *instantiation* via the `new` keyword. Another aspect which resembles classes as in Java is that a variables of class type contains either a pointer to an object of that class type or the special pointer value `null`.[1]

[1] Side remark: those class types and the corresponding objects are very simple. Without inheritance and methods and further "object-oriented complications", the class types resemble more closely to named record types than full classes —record types are otherwise also known as struct types— and objects correspond to *records* (also known as "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 ";" }  "end" ";"
DECL    -> VAR_DECL | PROC_DECL | CLASS_DECL
VAR_DECL    -> "var" NAME ":" TYPE
PROC_DECL    -> "proc" NAME
      "(" [ PARAM_DECL { "," PARAM_DECL } ] ")"
       [ ":" TYPE ]
      "begin" { DECL ";" } { STMT ";" } "end"
CLASS_DECL    -> "class" NAME "begin" { VAR_DECL ";" } "end"
PARAM_DECL    -> [ "ref" ] NAME ":" TYPE
EXP    -> EXP LOG_OP EXP
    | "not" EXP
    | EXP REL_OP EXP
    | EXP ARIT_OP EXP
    | "(" EXP ")"
    | LITERAL
    | CALL_STMT
    | "new" NAME
    | 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
IF_STMT    -> "if" EXP "then" "begin" { STMT ";" } "end"
[ "else" "begin" { STMT ";" } "end" ]
WHILE_STMT    -> "while" EXP "do" "begin" { STMT ";" } "end"
RETURN_STMT    -> "return" [ EXP ]
CALL_STMT    -> NAME "(" [ ACTUAL_PARAM { "," ACTUAL_PARAM } ] ")"
ACTUAL_PARAM    -> "ref" VAR | EXP
TYPE    -> "float" | "int" | "string" | "bool" | NAME
```

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 an "object")

4.3 Associativity

- The binary operations `||`, `&&`, `+`, `-`, `*`, and `.` are *left-associative*, but `#` are right-associative.
- Relation symbols are non-associative. That means that for example `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, the 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 two parameter passing mechanisms:

5.1 Call-by-value

This is the default. 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. This being the default, the parameters are just given without extra keywords specifying the parameter passing mechanism.

5.2 Call-by-reference

In this case, when calling a procedure the *address* of (or reference to) the actual parameter is passed into the formal parameter. Compila uses the keyword `ref`, which must be used in the procedure *definition* as well, in front of the corresponding actual parameters when calling the procedure.

5.3 Small example for call-by-reference

```
program SwapExample
  begin
    proc swap (ref a:  int, ref b : int) {
      var tmp : int;
      tmp := a;
      a := b;
      b := tmp;
  end;

  proc Main ( )
    begin
      var x : int;
      var y : int;
      x := 42;
      y := 84;
      swap (ref x,ref y);
      // now, x = 84 and y = 42
    end;
  end;
```

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 occurenccs of a name preceded by 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 class-type) and look up the field with name `NAME` in that class. It's an error, if `EXP` is not of class-type or else, there is not such field in that class,

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 class type.
  - the name standing after a dot are the name of a field/attribute of the class 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 class.

7.3 Types and implicit type coversion

It is allowed to assign an expression of type `int` to a variable of type `float`. The inverse situation is not allowed. There's 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 classes 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 class, 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.