Appendix **D**

Project (CC-only)

The project of Compiler Construction consists of the development of a complete compiler for a (source) language that you may define yourself. Read this appendix carefully to understand what is expected.

- §D.1 gives an introduction
- §D.2 describes the source language requirements
- §D.3 describes what you should do to test your compiler
- §D.4 describes the requirements for the end result (software and report)
- §D.5 describes how the project will be assessed

D.1 Introduction

The project consists of the following elements:

- *Source language:* To be defined by you. §D.2 describes the language features that should be supported.
- *Target language*: The default target language is ILOC, as described in EC, extended with some features that you have become acquainted with through the lab exercises. Documentation can be found on BLACKBOARD.

  Optionally, you may choose a different target language; for instance, JBC or .NET. This requires more effort on your side, but may be more satisfactory because these are languages used in practice. Also, choosing a different target language may improve your grade (see §D.5).
- *Compiler generator:* ANTLR version 4. The scanner and parser should be defined in ANTLR, and the rest of the compiler should be implemented through three listeners or tree visitors.
- *Implementation language:* JAVA in principle; however, ANTLR can generate other implementation languages (e.g., C# and PYTHONX, see `http://www.antlr.org/download.html`). You are free to use those instead, at the price of very limited assistance.

D.1.1 Global schedule for the project

- *Block 7:* Choice of language features, syntax definition, sample programs
- *Block 8:* Elaboration (type checking and other analyses)

priority	operators	valid operand types	result type
1	(unary) -, +	int	int
	!	bool	bool
2	*, /, %	int	int
3	+, -	int	int
4	<, <=, >=, >	int	bool
	==, <>	int, bool, char	bool
5	&&	bool	bool
6	||	bool	bool

Table D.1: Arithmetic operators, their relative priorities, and the types of their operands and result.

- *Block 9:* Code generation and testing
- *Block 10:* Report and submission
- *Final deadline:* Friday 14 July 2017. The rules for late submission apply.

During Blocks 7–10, there are regular scheduled hours of assistance. You will be asked to show your progress and discuss your choices at least once during this period.

D.2 Language features in the final project

This is based on section "Developing your own language" van [Henk Alblas, Han Groen, Albert Nymeyer and Christiaen Slot, Student Language Development Environment (The SLADE Companion Version 2.8), University of Twente, 1998].

In this section we describe a number of versions of a language based on the concept of *expressions*. Normally, we differentiate between statements and expressions: the former are used to carry out actions, and the latter to compute values. In the language that is proposed here, statements not only carry out actions, they also compute values. Moreover, declarations and statements may occur in any order. The only restriction is that the declaration of a variable or constant must precede its use.

While you may find suggestions for the syntax of this language in this chapter, *everyone is encouraged to choose his or her own syntax*.

We will use three types of data: integer, boolean and character. These data have an external representation on the keyboard and the screen, and an internal representation in memory. The boolean and character values are internally represented by integer values. The logical value `true` is internally represented by `1` and the value `false` by `0`. Characters are internally represented by their ordinal value in the ASCII-set. We begin by describing the basic expression language. We then describe two extensions to this language: a conditional statement and a while-statement. These extensions involve additional scope rules. We then present some more complicated extensions: procedures, functions, pointers (absolute addresses), arrays, and records.

D.2.1 Basic expression language (mandatory)

The basic expression language supports declarations and expressions. Declarations are either *constant* or *variable* declarations. An expression can be an arithmetic expression, an assignment statement, a read statement or a print statement. Every variable and constant must be declared, and the declaration of a variable or constant (called the defining occurrence) must precede its use (applied occurrence) in the text.

An *arithmetic expression* consists of a number of operands separated by operators. An operator can have the type integer (int), boolean (bool) and character (char). An operand can be a variable, constant or another expression, and also have the type int, bool and char. Examples of int, bool and char denotations are `12`, `true` and `'a'` (respectively). Not all types of operands, however, can be used in combination with all operators. The operators, their relative priorities (from highest to lowest), and the permitted combination of types is shown in Table D.1. Note that the operators == and <> are overloaded.

An *assignment statement* generates a result. This result is the value of the variable on the left-hand side of the assignment symbol. The type of an assignment statement is the type of its left-hand side variable. Further, the type of the left-hand side variable must be equal to the type of the right-hand side. Because an assignment statement has a value, it can be used as a 'sub-expression' in another statement or expression. Consider the following example:

```
x := y := x + y;
```

The value of the assignment statement `y:=x+y` is the value of `x+y`. This value is assigned to `x`, and is, therefore, also the value of the total statement. Notice that the operator `:=` is implicitly right-associative. An assignment statement cannot be used everywhere in an expression, however. The following expression, for example, is not allowed:

```
x + y := 1 + y;
```

Depending on their relative priorities, we could evaluate this expression as `x+(y:=1)+y`, which is `x+2`, or as `(x+y):=(1+y)`. Neither is desirable. We avoid this kind of construction by stipulating that there may only be a single variable on the left-hand side of an assignment operator.

A *read statement* has the general form `read(varlist)`, where `varlist` is a list of variables (at least one). A read statement also generates a result. The type of a read statement depends on what is read:

- If only a single variable is read, then the type of the read statement is equal to the type of this variable, and the result is its value.
- If more than one variable is read, then the read statement has type *void*.

A result that has type *void* corresponds to a value that cannot be used. More precisely, it corresponds to the empty value. The following expression, for example, is not allowed because the read statement has type *void*:

```
x + read(y, z)
```

A *print statement* has the general form `print(exprlist)`, where `exprlist` is a list of expressions (at least one). A print statement is analogous to a read statement, with the exception that not only can variables be printed, but also expressions.

- Each expression in a print statement must have a type that is not *void*.
- If only a single expression is printed, then the type of the print statement is equal to the type of this expression, and the result is its value.
- If more than one expression is printed, then the print statement itself has type *void*.

For example, the following statement prints the value of `x`, and then increments `x`:

```
x := print(x) + 1;
```

In contrast to a statement or expression, a declaration does not generate a result. A declaration is said to have type `no_type`.

The basic expression language also has a *compound expression*, which is a sequence of expressions and declarations, separated by semicolons. However, because a compound expression must also generate a result, we stipulate that a compound expression must end in an expression (and must not end in a declaration). The result and type, then, of the compound expression is the result and type of this (final) expression. The scope of any declaration in the compound expression is the compound expression itself, but declaration must precede use. Consider, for example, the compound expression that reads two boolean variables and evaluates a boolean expression:

```
var a: boolean; read(a);
var b: boolean; read(b);
(a && !b) || (!a && b);
```

This compound expression has type boolean, and generates the *exclusive-or* of the two variables. Note that the syntax that we have used here for the declaration is only a suggestion. We could also have written the declarations using the form `var boolean a`, for example, or even `boolean a`.

A compound expression that is enclosed within an open- and close symbol (e.g. using curly brace: '{' and '}', or `begin` and `end`, etc), is called a *closed compound expression*. The result and type of a closed

compound expression is the same as the result and type of the enclosed compound expression. Because a closed compound expression generates a result, it can also be an operand. For example, we can assign the result of the above compound expression to some boolean variable `c` as follows:

```
c := { var a: boolean; read(a);
       var b: boolean; read(b);
       (a && !b) || (!a && b);
     } ;
```

In this example, we used the curly braces '{' and '}' characters to enclose the compound expression into a closed compound expression, but the actual choice of syntax is up to yourself. Further note that the *boolean* variables `a` and `b` are only defined inside the closed compound expression.

To understand the consequences of treating statements as expressions, particularly from an implementation point of view, let us look at some program constructs. Consider the following program fragment:

```
x := y:= 1; z := 2;
```

The result of the assignment statement `y:=1` is the value `1`. This value is then assigned to `x`. In a sense, this value is being re-used. The assignment to `x` also generates a value `1`. This value, however, is redundant and must be discarded. The following assignment statement (`z:=2`) is then executed, and generates the value `2`. Depending on the context of this fragment, this value may also have to be discarded. In practice, when an expression is executed, it leaves a value on the arithmetic stack, ready to be used by another expression. If this value is not used, then it must be popped off the stack. This situation arises when we have two expressions separated by a semicolon. [1]

We will see that there are other situations where values need to be discarded. The value generated by the last expression in the main program, for example, must also be discarded (the program does not generate a result). Consider, for example, the following program:

```
begin
    x := 1;
    print(x);
end.
```

When the `print`-statement is executed, it generates the value of `x`, namely `1`. This value must be discarded. Care should be taken in building the compiler to ensure that values that are generated by expressions are either re-used or explicitly discarded.

D.2.2 Conditional statement (mandatory)

We can extend and improve the basic expression language by adding a conditional statement. A conditional statement adds more expressive power to the language. Like assignment, read and print statements, a conditional statement generates a result. We can use it in the following way, for example:

```
x := if b then 0 else 1 fi;
```

Depending on the value of `b`, `x` will be set to `0` or `1`. Because a conditional statement generates a result, it can be used as an operand. Possible operands, then, in the extended expression language are conditional statements, closed compound expressions, and variables, constants and denotations of type integer, boolean and character. The general form of a conditional statement is as follows:

`if` $expr_0$ `then` $expr_1$ `else` $expr_2$ `fi`

where each $expr_i$ is a compound expression. Note that this is only a suggested syntax. The else part (i.e., `else` $expr_2$) in a conditional statement is optional. The following conditions on the types apply:

- The type of $expr_0$ must be boolean.
- If there is no `else` part, then the statement has type *void*.
- If there is an `else` part, then:
  - If $expr_1$ and $expr_2$ have the same type, then this is the type of the conditional statement.

[1] Instead of *always* leaving a value on the arithmetic stack (and subsequently discarding the value by popping the value), the code generator could also be directed to *only* leave a value on the stack when this value is actually needed. This is much more elegant and leads to more efficient target code.

– If $expr_1$ and $expr_2$ have different types, then the conditional statement has type *void*.

Further, the following special scope rules apply.

- The scope of declarations in $expr_0$ is all three compound expressions.
- The scope of declarations in $expr_1$ is only itself.
- The scope of declarations in $expr_2$ is only itself.

D.2.3 While statement (mandatory)

An iterative construct is added to the expression language in the form of a *while statement*. A `while` statement has the general form:

`while` $expr_0$ `do` $expr_1$ `od`

The following conditions on the type and scope apply:

- The while statement has type *void*.
- The type of $expr_0$ is boolean.
- The scope of declarations in $expr_0$ is both compound expressions.
- The scope of declarations in $expr_1$ is only itself.

Because a while statement has type *void*, it cannot be used as an operand. In fact, a while statement is an expression, along with assignment statements, read statements and print statements. Consider the following example of a while statement:

```
while b do x := x + 1; print(x) od;
```

The `print` statement within the `while` statement generates a result. This result must be discarded because the `while` statement has type *void*.

D.2.4 Procedures and functions (optional)

The body of a *procedure* is a closed compound expression that has type *void*. This means that a procedure cannot be an operand. A procedure may have value parameters and reference (= `var`-) parameters. Both kinds of parameters are, of course, operands, and can be of type integer, boolean and character. Recursive calls should be supported.

An example of a declaration and call of a *procedure* is illustrated the following program:

```
begin
    var a, b: integer;
    procedure swap(var x, y: integer) {
        var z: integer;
        z := x; x := y; y := z;
    }
    a := 1; b := 2;
    swap(a, b);
    print(a, b);
end.
```

A *function* is similar to a procedure. The differences are:

- A function call is an operand.
- The closed compound expression that is the body of the function generates a value. This is the value returned by the function. The type of this value (either integer, boolean or character) is also the type of the function.

Recursive function calls should be supported.

An example of a declaration and call of a *function* is illustrated in the following program:

```
begin
    function fac(n: integer): integer {
        return n * fac(n-1);
    };
    print(fac(10));
end.
```

D.2.5 Arrays (optional)

`Arrays` allow data to be conveniently structured. For simplicity, we only consider 1 and 2-dimensional arrays. To declare and use arrays, we add the following language constructs. Note that the syntax used in the examples shown below is for illustrative purposes only.

- *type declaration.* Array types allow a new array data type to be defined. In the type definition, bounds are placed on the indices. These bounds must be integer denotations. For example:

  ```
  type barray = array [1..4] of boolean;
  type iarray = array [1..2, 1..2] of integer;
  ```

  Note that, because the bounds are integers, the bounds can always be statically determined.

- *variable declaration.* Array variables can be declared by using the array type. For example:

  ```
  var b: barray;
  var x, y: iarray;
  ```

- *variable.* Array variables and constants can be used in the program text. Arrays can be assigned (using the assignment operator `:=`), and they can be compared (the operators `==` and `<>`). For example:

  ```
  if x <> y then x := y fi;
  ```

  where `x` and `y` have been declared above as having an array type.

- *indexed variable.* Indexed array variables can be used to access elements in an array. An index is an integer expression, and is traditionally enclosed in square brackets. For example:

  ```
  i := 1; x[i] := y[i];
  ```

- *denotation.* Array denotations are also possible. For example:

  ```
  b := [true, false, true, false];
  x := [[7, 1], [7, 31]];
  ```

  will initialise the two arrays declared above.

Array variables and indexed array variables can be used as operands. In the case of array variables, however, only the operators `==` and `<>` can be used. Indexed array variables have the type integer, boolean and character, and therefore satisfy Table D.1.

Constant array declarations. Just as 'variables' can be declared as constant, it should also be possible to declare array variables as constant. Consider, for example, the following declaration:

```
const a: barray = [true, false, true, true];
```

Note that the implementation of this construct, however, is more difficult than other array constructs.

D.2.6 Records (optional)

Records can be seen as a generalisation of arrays. The specification of records, therefore, follows the same lines as arrays. It involves adding a record type declaration, record variable declaration, record variables, record field variables (which are analogous to indexed arrays) and record denotations. A record consists of *fields*, and these fields can be of type integer, boolean and character. Below we describe how records are declared and used. As with arrays, the syntax used in the examples is for illustrative purposes only.

- *type declaration.* Record type declarations define the structure of a record. A record consists of a number of field variables. Each field variable has a certain type. For example:

```
type mix = record [ a: integer; b: boolean; c: character; ] ;
```

  In this record type declaration, three field variables have been declared.

- *variable declaration.* Record variables can be declared by using the previously declared record type. For example:

```
var r, s: mix;
```

- *variable.* Record variables and constants can be used in the program text. Records can be assigned (:=) and they can be compared (== and <>). For example:

```
if r <> s then r := s fi;
```

- *field variable.* The record field variables are identified by a record and field name, separated by a dot. For example:

```
r.a := 1; r.b := false; r.c := 'a';
```

- *denotation.* Record denotations can be used to initialise a record. A record denotation consists of the record type, followed by the contents of the fields, and surrounded by square brackets. For example:

```
r := [1, true, 'a'] ;
```

Like variables, constants and denotations, therefore, record variables and field variables are operands. However, only the operators == and <> can be used with record variables. Because field variables can only have the types integer, boolean and character, field variables satisfy Table D.1.

Constant record declarations. As an optional extra, we could also consider constant record declarations. Take, for example, the following declaration.

```
const r: mix = [1, true, 'a'] ;
```

However, analogous to arrays, the implementation of this construct is more difficult than other record constructs.

D.2.7 Pointers (optional)

We now add *pointers* to our expression language. A pointer has a value, which is the address of a variable, or `nil`. When a pointer is declared, we must specify the type of the variable to which it points. For example:

```
var p, q: pointer to integer;
```

Pointers are assigned by using an address function, e.g., `address(v)`, where `v` is a variable. For example:

```
p := address(i);
```

The inverse of this function, e.g., `value(p)`, yields the value pointed to by pointer `p`. For example:

```
i := value(p);
```

where, in this case, the variable `i` would have to be declared as an integer. The value `nil` can also be assigned to a pointer, as in:

```
q := nil;
```

Note that `nil` has no single type, `nil` can be assigned to a pointer of any type. Like arrays and records, pointers are variables that can be assigned (e.g. `p := q`) and compared (e.g. `p == q` and `p <> q`). Note that a pointer to a variable is only valid as long as the variable is within its scope. A pointer to a variable that has 'ceased to exist' is called a *dangling pointer*. The compiler must check that pointers do not dangle. Consider, for example, the statement:

```
p := (var i: integer; i := 17; address(i));
```

The value of the closed compound expression is the address of `i`. The pointer dangles because `i` is not defined outside the closed compound expression. In general, globally-declared pointers that point to local variables must not be used outside the scope of these variables.

D.3 Testing

This is based on chapter "Self-testing your compiler" van [Henk Alblas, Han Groen, Albert Nymeyer and Christiaen Slot, Student Language Development Environment (The SLADE Companion Version 2.8), University of Twente, 1998].

A compiler behaves correctly if every target program conforms to the language specification. Instead of fully verifying our compiler, we can increase our confidence in its correctness by applying a series of tests. Note that testing can only ever show the presence of errors, not their absence. Nevertheless, careful testing is useful and necessary in situations where full, formal verification is not possible. The advantage of testing is that it can be carried out independent of the way the compiler is specified and constructed. Ideally, the tests should consist of all possible programs. Unfortunately, in most languages an infinite number of programs can be written, so all we can hope to do is to judiciously select a subset of all programs, called a test set, that is in some way representative of the language. A test set should not only contain correct programs, but also programs that contain errors, so that we can see how the compiler handles incorrect input. Errors in a program can occur in the:

- lexical syntax (e.g. spelling errors)
- context-free syntax (e.g. language-construct errors)
- context constraints (e.g. declaration, scope and type errors)
- semantics (e.g. run-time errors)

In the next section we will discuss a test set for the basic expression language (see §D.2.1).[2] By studying this test set you will be able to build an extensive test set for your own language. In subsequent sections we will discuss the construction of test sets for each of the different extensions to the basic expression language.

D.3.1 Basic expression language

For the basic expression language of §D.2.1 we will construct 4 test programs:

- a correct test program
- a test program containing spelling and context-free syntax errors
- a test program that violates the context constraints
- a test program with a run-time error

A correct test program. To test the compiler at the level of syntax a test program must be written that contains identifiers and denotations of integers and characters, all possible keywords and symbols, and the booleans `true` and `false`. We will include these in a test program in which also context-free syntax constructs (in the basic expression language these are the declarations and expressions) are checked. We consider first the declarations and then the expressions, and we distinguish between arithmetic expressions, assignment statements, read statements, print statements and compound expressions.

Declarations. Every variable and constant must be declared with type integer, boolean or character. A test program should therefore contain:

```
var ivar1, ivar2: integer;
var vvar: boolean;
var cvar1, cvar2: character;
const iconst1: integer = 1, iconst2: integer = 2;
const bconst: boolean = true;
const cconst: character = 'c';
```

All variables and constants used in a program must be declared. We can use the above-mentioned declarations in a larger test program to check this. We will do this later.

Operators and operands. Table D.1 shows the operators, their relative priorities, and the operand and result types. The following expression is a test of the relative priorities of `+`, `-` and `*`, for example.

```
+16 + 2 * -8
```

[2] Again, the syntax of the language as used in this section just an example. You are encouraged to choose your own terminals and symbols.

```
begin
    var ivar: integer;
    ivar :=
        {
            var ivar1, ivar2: integer;
            read(ivar1, ivar2);
            write(ivar1, ivar2);
            const iconst1: integer = 1;
            const iconst2: integer = 2;
            ivar2 := ivar1 := +16 + 2 * -8;
            write(ivar1 < ivar2 && iconst1 <= iconst2,
                  iconst1 * iconst2 > ivar2 - ivar1);
            ivar1 < read(ivar2) && iconst1 <= iconst2;
            ivar2 := write(ivar2) + 1;
        } + 1;
    var bvar: boolean;
    bvar :=
        {
            var bvar: boolean;
            read(bvar);
            write(bvar);
            bvar := 12 / 5 * 5 + 12 % 5 = 12 && 6 >= 6;
            const bconst: boolean = true;
            write(!false && bvar == bconst || true <> false);
        } && true;
    var cvar: character;
    cvar :=
        {
            var cvar1, cvar2: character;
            read(cvar1);
            const cconst: character = 'c';
            cvar2 := 'z';
            write('a', cvar1 == cconst && (cvar2 <> 'b' || !true));
            'b';
        };
    write(ivar, bvar, cvar);
end.
```

Figure D.1: Test program that checks for correct syntax.

Expressions with a boolean result type can be built using operands of different types. For example:

```
12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6
ivar1 < ivar2 && iconst1 <= iconst2
iconst1 * iconst2 > ivar2 - ivar1
cvar1 = const && (cvar2 <> 'b' || !true)
!false && bvar = bconst || true <> false
```

Note that in these expressions we use all operators and all possible operand types. Finally, a simple character expression:

```
'b'
```

Assignments. There can be simple and multiple assignment statements. For example:

```
ivar2 := ivar1 := +16 + 2 * -8;
bvar  := 12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6;
cvar2 := 'z';
```

Read and print. A read statement reads a list of values of variables. Some examples are:

```
read(ivar1, ivar2);
read(bvar);
read(cvar1);
```

A print statement can be more complicated as whole expressions must be handled. For example:

```
write(ivar2);
```

```
begin
    var a, b: integer;

    // an error in an expression:
    a + * b;

    // an incomplete assignment token:
    a :-b;

    // non-existing and misspelled keywords:
    for gebin ned repeat;

    // hurray, finally something good:
    a + b;
END.
```

Figure D.2: A test program that contains spelling and syntax errors.

```
write(bvar);
write(!false && bvar == bconst || true <> false);
write(ivar1, ivar2);
write(ivar1 < ivar2 && iconst1 <= iconst2,
      iconst1 * iconst2 > ivar2 - ivar1);
write('c');
write('a', cvar1 = cconst && (cvar2 <> 'b' || !true));
```

In the following example we check that read and print statements can also occur as operands in an expression.

```
ivar1 < read(ivar2) && iconst1 <= iconst2;
ivar2 := write(ivar2) + 1;
```

Compound expressions. We now consider compound expressions, which are sequences of expressions and declarations, separated (or terminated) by semicolons. Declarations and expressions may occur in any order as long as declarations of variables and constants always precede their use, and the compound expression ends in an expression.

In Figure D.1 we show our first test program. It contains three compound expressions composed from the language constructs that we have discussed so far. The program consists of three assignment statements. The right-hand side of each assignment is a closed compound expression, and each closed compound expression introduces a scope and delivers a value.

Note that we use the symbols '{' and '}' to enclose a compound expression. We could have used other syntax here instead (e.g. using the keywords `begin` and `end`).
Sample input for this program is: `0 1 1 false c`,
which generates the output: `0 1 false true 1 false true a true 3 true b`.

A test program containing spelling and context-free syntax errors. In the initial stage of program development simple syntax errors occur frequently. These range from spelling mistakes to incorrect program constructs. The scanner and parser generator should provide for error recovery, i.e., they add special functions to the generated scanner and parser to detect and recover from these errors. To see how the generated compiler handles these errors we could use the small incorrect program shown in Figure D.2. Note that Java-style line comments are used are used to specify comments (i.e., `//` ...).

A test program that violates the context constraints. This can be a difficult source for errors because errors in the context constraints are usually concerned with the actions in the elaboration phase. We highlight here a few of the more common error conditions, and give a test program to test for these errors.

Incorrect assignments. There may only be a single variable on the left-hand side of an assignment operator, and constants, denotations and expressions are not allowed. Some incorrect assignments are:

```
var x, y: integer;
const z: integer = 1;
z := 10; 12 := 10;
x + y := 10;
```

Type errors. Operators must be applied to operands of the correct type. Below we present some incorrect combinations.

```
- 'a';
+ true;
var c: character;
!c;
var b: boolean;
b + 10 * c % 2;
'a' < c;
b && 10 || c;
```

De binary operators == and <> are *overloaded*, i.e., they may be applied to integers, booleans and characters, but both operands of these operators must be of the same type. The following combinations are therefore not allowed (see the aforementioned declarations).

```
'a' == b;
b <> 10;
x + y == c;
```

The left and right-hand sides of an assignment must be of the same type. The following assignments are therefore incorrect.

```
c := x + y;
b := 10;
```

Missing declarations. The declaration of a variable or constant must precede its use. In the following program fragment the variable or constant p is not declared, and the declaration of q comes too late.

```
p; q;
var q: boolean;
```

A compound expression without a result. Declarations and expressions may occur in any order. However, because a compound expression must generate a result, a compound expression must end in an expression. The following closed compound expression ends in a declaration, and is therefore incorrect.

```
{ 2 + 4 * 3; var w: integer; }
```

Operations on operands of type void. Operators may not be applied to operands of type *void*. The following constructs are therefore not allowed.

```
read(x, y) + 10;
'c' <> write(x, y);
10 + { var u, v: integer; read(u, v); write(u, v); };
```

Note that in the last line of the program, the statement `write(u, v)` delivers a result of type *void*, and as a result, the closed compound expression is of type *void*. We now combine all the above erroneous constructs into one test program that checks for violation of context constraints. We show this program in Figure D.3. *Note, however, that it usual more convenient to store the individual tests in small test programs to ease the unit testing of your compiler.*

A test program with a run-time error. We complete our test set with a program that contains a run-time error. In the case of simple languages (i.e., those without conditional and repetitive statements, procedures, functions, and structured variables), the context analyzer could check at compile time whether all variables appearing in an expression have been assigned. This means that division by 0 (or using the `MOD`-operator) is the only thing that might go wrong during the execution of a program in the basic expression language. The program in Figure D.4 exhibits this run-time error. Other run-time errors can occur in more involved languages. For example, the use of non-assigned variables, index of an array out of bounds, and reference to a non-initialised or null pointer.

```
begin
    var x, y: integer;
    const z: integer = 1;
    z := 10;
    12 := 10;
    x + y := 10;

    - 'a';
    + true;
    var c: character;
    !c;
    var b: boolean;
    b + 10 * c % 2;
    'a' < c;
    b && 10 || c;

    'a' = b;
    b <> 10;
    x + y = c;

    c := x + y;
    b := 10;

    p; q;
    var q: boolean;

    { 2 + 4 * 3; var w: integer; };

    read(x, y) + 10;
    'c' <> write(x, y);
    10 + { var u, v: integer; read(u, v); write(u, v); };
end.
```

Figure D.3: Test program that checks context constraints.

```
begin
    10 / 0
end.
```

Figure D.4: Test program that contains a run-time error.

D.3.2 Conditional statement

If the basic expression language is extended with a conditional statement, then some tests need to be developed that check combinations of the conditional statement and other constructs of the basic expression language. Because of the special type conditions, the use of a conditional statement as an operand requires special attention. Furthermore, the scope rules that apply to conditional statements must be checked.

D.3.3 While statement

The extension of the basic expression language with a while statement requires similar tests on the type and scope. The fact that a while statement has *void*, and thus cannot be used as an operand, requires special attention. Furthermore, the scope rules that apply to the boolean expression should be taken into account.

D.3.4 Procedures and functions

For a language with procedures we again need more test programs. The body of a procedure is a closed compound expression. The tests that we applied to a closed compound expression can therefore also be used to check procedure bodies. However, because a procedure body is of type *void*, a procedure cannot be used as an operand.

The concept of a procedure introduces new scope rules. These scope rules will require extra tests to check the visibility of variables and constants. The visibility of a procedure name is similar to the visibility

of a variable, i.e., a procedure may only be called within the scope of its declaration, and a procedure must be declared before it is called. The scope of a parameter is the block of the procedure declaration.

The correspondence between the arguments of the call statement and the parameters of the procedure declaration (i.e., the number and types of the arguments) is another aspect that needs to be tested. Special attention should be paid to the correct use of value and reference (= `var`-) parameters. In the case of a value parameter the argument must be an expression, and in the case of a reference parameter, the argument must be a variable or a reference parameter from a surrounding procedure. Finally, note that a procedure can call itself recursively.

Functions are similar to procedures and thus require similar tests. The main differences are that a function call is an operand, and the body of a function generates a return value, which must be of the same type as the function.

D.3.5 Arrays

Adding arrays to the the basic expression language requires test programs that check whether array types can be defined, and array variables can be declared by using these array types. Moreover, indexed array variables can be used to access elements in an array, and these indexed variables can be assigned and used in expressions.

The use of array indices requires bound checking at run-time. This means that run-time tests are needed to check if the bound-checking algorithm of the compiler is correct. Special tests are required for array variables and constants (denotations) because complete arrays can be assigned and compared.

D.3.6 Records

Records are a generalisation of arrays and require similar tests. However, the definition of field variables and the form of their assignment and use are different.

D.3.7 Pointers

In a sense, pointers are similar to variables. They need to be declared before use, they point to variables of a certain type, and they can be used and assigned. The difference is that they have in fact two values, a direct value (an address value or `nil`) and an indirect value (the value of the variable pointed to). A test set for pointers should test the address function, its inverse, the assignment and comparison of address values, and the assignment and use of the value `nil`. A pointer to a variable is only valid as long as the variable is within its scope. Special tests are needed to check how the compiler handles dangling pointers.

D.4 The final product

The product you should submit for the final project consists the developed software and a printable report. These should be uploaded to BLACKBOARD in a single zip-file. The general guidelines for late submission apply.

D.4.1 Software

The submitted zip-file should contain the following elements (in a well-structured directory hierarchy):

- *The report*, in PDF-format.
- *A `README`-file* with instructions using the compiler. Such instructions may include, but are not limited to, an overview of the directories and files necessary for execution and the required steps for installation and invocation. Upon following these instructions, an end user should be able to obtain and invoke use a working compiler. *If this requirement is not met, the submission will not be graded; non-compiling programs are not accepted.*
- *Full* ANTLR *grammars* as well as the JAVA files generated therefrom.

- *Source code* of all classes programmed by you, in a single class hierarchy. The code should meet the following criteria:
  - Compiles without errors*
  - Contains documentation (in the form of JAVADOC)*
  - Meets common coding standards, for instance regarding naming, package structure, and visibility of fields.

  The criteria marked * are necessary to score more than 4,0 for the project.

- *Bytecode* of all predefined classes and libraries, insofar they are not part of the standard JAVA runtime environment.

- *Results of all tests.* For *correct* test programs, the result should consist of
  - The source code of the program itself
  - The generated ILOC code
  - Some test runs

  For *incorrect* test programs, the result shoud consist of
  - The source code of the program itself
  - The output generated by the compiler for the program (i.e., the error messages)

Again, take care that your code compiles and runs after following the instructions in the enclosed `README`-file. *We should not have to change anything in your source code!* Typical cases where this goes wrong are: names and paths of files or other URLs, like host machines and servers. *Test this before submission.*

D.4.2 Report

The report should give insight in how the language has been defined, and how any problems occurring during the construction of the compiler were solved. The report should contain the following parts; for each part, list who of you was responsible, or whether the responsibility was shared equally.

- *Front page.* Clearly list the authors, including (for each student) first and last name and student number.

- *Summary* of the main features of your programming language (max. 1 page).

- *Problems and solutions.* Summary of problems you encountered and how you solved them (max. two pages).

- *Detailed language description.* A systematic description of the features of your language, for each feature specifying
  - Syntax, including one or more examples;
  - Usage: how should the feature be used? Are there any typing or other restrictions?
  - Semantics: what does the feature do? How will it be executed?
  - Code generation: what kind of target code is generated for the feature?

  You may make use of your ANTLR grammar as a basis for this description, but note that not every rule necessarily corresponds to a language feature.

- *Description of the software:* Summary of the JAVA classes you implemented; for instance, for symbol table management, type checking, code generation, error handling, etc. In your description, rely on the concepts and terminology you learned during the course, such as synthesised and inherited attributes, tree listeners and visitors.

- *Test plan and results.* Discussion of the correctness test, using the criteria described in §D.3. You should provide a set of test programs demonstrating the correct functioning of your compiler. The test set should contain, next to programs testing the various language featurs, also programs containing syntactic, semantic or run-time errors.

  All tests should be provided as part of the zip-file. One test program should be included as an appendix in the report (see below).

- *Conclusions*.

Appendices. In addition to the above, your report should also contain the following appendices:

- ANTLR *grammar(s).* The complete listing of your ANTLR grammar (or grammars if you have split it up).

- *All* ANTLR *tree walkers (listeners and visitors).* The complete listing of each implementation of a tree listener or tree visitor for your grammar.

- *Extended test program.* The listing of one (correct) extended test program, as well as the generated target code for that program and one or more example executions showing the correct functioning of the generated code.

Check the readability of your listings, if necessary by putting them into landscape mode. If you used tabs, make sure the tab stops are the same for your editor and your printout.

(The reason to include listings in your report of files that are also provided in your zip-file is primarily to make it easier to assess your work. The idea is that the report is the basis of the assessment; ideally, it should not be necessary to study your code separately.)

D.5 Assessment

The grade for the final assignment depends on:

- The language constructions supported in your programming language and compiler;
- The target language (ILOC, JBC, .NET, LLVM);
- The quality of the implementation;
- The report

§D.2 describes the minimally required language features; in particular, all languages should at least satisfy the requirements of the "Basic Expression Language" (§D.2.1).

Table D.2 shows how the *basic grade* of the final project is determined. This shows that translating to some other target language than ILOC yields one bonus point. Possible langugage extensions should be added in the order listed in Table D.2.

In addition, you may consider providing extra functionality in your language and compiler. Table D.3 shows how such extensions are graded. Note that you can theoretically achieve an overall grade of more than 10,0 in this way; however, in such a case your grade will be rounded down. (But you will have learned a lot!)

Extensions such as **`switch`**-statements, **`for`**-statements or **`repeat/until`**-statements are regarded as "more of the same" and do not yield extra points; however, they can be used as reasons to round the final grade up.

The definitive grade for the final project furthermore depends on:

- Structure of the ANTLR grammar: `-1 ... +1`
- Structure and documentation of the JAVA code: `-2 ... +1`
- Quality of the report: `-2 ... +1`
- Quality of the tests: `-1.5 ... +1.5`

language feature	ILOC	JBC	.NET
basic expression language	6.0	7.0	7.0
+ `if` and `while`	6.5	7.5	7.5
+ procedures and functions	7.5	8.5	8.5
+ arrays	8.5	9.5	9.5

Table D.2: Compiler Construction assessment – **basic grade**.

extension	ILOC	JBC	.NET
+ enumerated types	+0.25	+0.25	+0.25
+ records	+0.5	+0.5	+0.5
+ pointers	+0.5	-	+0.5
+ strings	+0.5	-	-
+ exception handling	+1.0	+0.5	+0.5
+ dynamic objects, free/delete	+1.5	+0.5	+0.5
+ object orientation (classes)	+1.5	+0.5	+0.5

Table D.3: Compiler Construction assessment – **extensions**.

Please follow the language feature descriptions in §D.2. Simplifying the project will lead to a reduced grade.

Examples.

- Two students implement a compiler for the basic expression language with **`if`/`while`**, compiling to JAVA byte code (JBC). In addition, strings are added. The compiler has unfortunately been tested very poorly, resulting in a 1 point deduction. The resulting grade is: $7.5 + 0.0 - 1.0 = 6.5$.
- Two students implement a compiler for the basic expression language, with ILOC as target language. They also support enumerated types and records. The report are tests are very good, resulting in a bonus point. The resulting grade is: $6.0 + 0.25 + 0.25 + 1.0 = 7.5$.

D.5.1 Feel like a challenge?

If you do find this project challenging enough or the requirements unnecessarily restrictive, if you do not want to be constrained by the imperative straightjacket or you want to compile to a different target architecture (for instance, LLVM), this is allowed; however, make sure you contact the teacher in time.