Bijlage A

Eindopdracht

PRACTICUM

VB 2010/2011 – Eindopdracht. Tijdens het tweede practische deel van het vak Vertalerbouw dient een complete vertaler te worden ontwikkeld voor een zelf te definiëren programmeertaal. Lees deze appendix aandachtig door om te weten wat er van u verwacht wordt. §A.1 geeft een inleiding op de eindopdracht. §A.2 geeft aan hoe de eindopdracht beoordeeld zal worden. In §A.3 wordt uitgezet waaraan het verslag van de eindopdracht moet voldoen. De laatste twee secties zijn in het Engels: in §A.4 worden de omschrijvingen van de taalfeatures gegeven en in §A.5 wordt uiteengezet hoe de compiler getest dient te worden.

A.1 Inleiding

Tijdens de eindopdracht van het vak Vertalerbouw wordt een eigen programmeertaal gedefinieerd en de bijbehorende vertaler gebouwd. Er wordt daarbij gebruik gemaakt van de compiler-generator ANTLR, die in de weken 3 en 4 van het practicum ook al voor de eenvoudige taal Calc gebruikt is. Daarnaast zal er een verslag van de taal en de ontwikkeling van de vertaler worden geschreven. De eindopdracht wordt beoordeeld op de kwaliteit van de ontwikkelde programmatuur en het eindverslag. Zie §A.2 voor details.

A.1.1 Randvoorwaarden

Randvoorwaarden voor het ontwikkelen van de compiler zijn de volgende:

- *Brontaal:* zelf te definiëren. Sectie §A.4 geeft de eisen waaraan uw taal(constructies) tenminste moeten voldoen.

- *Targettaal:* (in principe) Triangle TAM's assembler language. U dient hierbij de in week 4 geïntroduceerde `TAM.Assembler` te gebruiken om TAM-programma's naar TAM-bytecode te converteren. Als extra uitdaging zou u ook naar de Java Virtual Machine of naar .NET kunnen compileren; §A.2 geeft aan hoeveel punten dit extra zou kunnen opleveren.
- *Compiler generator:* ANTLR. U dient ANTLR versie 3.2 te gebruiken[1] voor het genereren van (i) de scanner, (ii) de parser, (iii) de context analyser (als treewalker) en (iv) de code generator (als treewalker).
- *Implementatie taal:* Java (tenminste versie 5 aka 1.5). Echter, omdat ANTLR ook C++ en C# kan genereren is het ook toegestaan om C++ of C# als implementatietaal te gebruiken. Er wordt echter geen ondersteuning geboden voor andere programmeertalen dan Java.

Zorg ervoor dat de ANTLR specificaties zo 'puur mogelijk blijven' en zo weinig mogelijk Java definities bevatten. Alle Java programmatuur die u nodig heeft om uw compiler te implementeren (b.v. voor de symbol table, context checks, code generatie, error handling, etc.) dienen in aparte klassen en/of packages terecht te komen. Van deze Java programmatuur hoeft alleen een beknopte beschrijving in het verslag te worden opgenomen. De complete programmatuur dient wel op een CD-R worden bijgevoegd.

Als uitgangspunt van uw vertaler zou u naar uw eigen compiler van de Calc-programmeertaal kunnnen kijken. Let echter wel op: de expressietaal van de eindopdracht verschilt op wezenlijke punten van het simpele Calc-taaltje.

A.1.2 Globale indeling weken 7 t/m 12

Tijdens de verroosterde practicumuren op woensdagmiddag zijn er studentassistenten aanwezig voor de begeleiding van de eindopdrachten. Hieronder staat een richtlijn voor de planning van de implementatie van uw compiler tijdens het tweede deel.

- kw4 **wk7** *(2011: week 22)*: scanner en parser
  - definitie van de programmeertaal: syntax (EBNF grammatica), context-beperkingen en semantiek;
  - scanner (= lexer) specificatie in ANTLR;
  - parser specificatie in ANTLR;
  - voorbeeld- en testprogramma's opstellen (zie ook §A.5).
  - voorbeeldprogramma's testen (met name de gegenereerde ASTs) met de gegenereerde scanner en parser;
- kw4 **wk8** *(2011: week 23)*: contextchecker
  - bibliotheek voor de symbol table en acties voor het controleren van context beperkingen.
  - context checking: specificatie van een treeparser in ANTLR die de context regels van de AST controleert; deze pass voegt ook identifier informatie aan de AST nodes toe;
  - context-beperkingen testen met de voorbeeldprogramma's.
- kw4 **wk9** *(2011: week 24)*: codegenerator

[1] Het is *niet* toegestaan om oudere versies van ANTLR te gebruiken.

taalelementen	*TAM*	*JVM*	*.NET*
basic expression language	6.0	7.0	7.0
+ `if` en `while`	6.5	7.5	7.5
+ procedures en functies	7.5	8.5	8.5
+ arrays	8.5	9.5	9.5

Tabel A.1: Beoordeling Vertalerbouw – **basiscijfer**.

- code generatie: specificatie van een treeparser in ANTLR die target code genereert gegeven de AST gegenereerd door de context checker;
- code generatie testen met de voorbeeldprogramma's

Week 9 is de laatste week dat er begeleiding bij het practicum is.

- kw4 **wk10** *(2011: week 25)*: verslaglegging.
- kw4 **wk11 en wk12** *(2011: weken 26 en 27)*: eventuele uitloop.

De deadline voor het eindproduct van de eindopdracht Vertalerbouw is **woensdag 6 juli 2011**.

A.2 Beoordeling

Het cijfer voor de eindopdracht Vertalerbouw hangt af van (i) de taalconstructies die ondersteund worden in de gedefinieerde taal en de bijbehorende compiler, (ii) de gebruikte doeltaal (TAM, JVM of .NET), (iii) de kwaliteit van de programmatuur en (iv) het verslag. In §A.4 worden de randvoorwaarden van de taalconstructies besproken. *Alle talen moeten ten minste aan de eisen van de 'Basic Expression Language' van §A.4.1 voldoen.*

In Tabel A.1 staat vermeld hoe het *basiscijfer* voor de eindopdracht Vertalerbouw wordt opgebouwd; vertalen naar JVM of .NET levert dus één punt extra op. De mogelijke uitbreidingen op de expressietaal dienen in de volgorde van Tabel A.1 geïmplementeerd te worden.

Daarnaast zou u kunnen overwegen om extra functionaliteit in uw taal en compiler in te bouwen. Tabel A.2 geeft een overzicht van extra's die aan de compiler zouden kunnen worden toegevoegd en de bijbehorende waardering in punten. Hoewel het cijfer met deze extra's theoretisch boven de 10.0 zou kunnen komen, zal dit in dit praktijk maximaal 10.0 kunnen zijn. Zij opgemerkt dat sommige van de uitbreidingen van Tabel A.2 niet eenvoudig zijn; uw studentassistent zal niet alle vragen kunnen beantwoorden. Aan de andere kant kunnen sommige uitbreidingen een extra dimensie aan het practicum Vertalerbouw geven. Extra's als `case`-statement, `for`-statement, `repeat`/`until`-statement worden daarentegen beschouwd als 'variaties op een thema' en leveren geen extra punten op, maar kunnen wel gebruikt worden voor afronding.

Het definitieve cijfer voor de eindopdracht hangt verder af van:

- opbouw grammatica en ANTLR specificatie: `-1` ... `+1`
- opbouw Java programmatuur en commentaar: `-2` ... `+1`
- kwaliteit van het verslag: `-2` ... `+1`
- kwaliteit van de tests: `-1.5` ... `+1.5`

uitbreiding	*TAM*	*JVM*	*.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 orientatie (classes)	+1.5	+0.5	+0.5

Tabel A.2: Beoordeling Vertalerbouw – **extra functionaliteit**.

- voor de ANTLR LL(k) parser moet gelden gelden dat `k=1`; als `k>=2`, dan kost dit minstens 1 punt op het cijfer voor de eindopdracht.

Zorg dat u zoveel mogelijk de beschrijving van de taal-gedeelten uit §A.4 volgt. Afwijkingen die de opdracht *vereenvoudigen* worden streng aangerekend.

Voorbeelden.

- Een tweetal studenten implementeert een compiler voor de expressie-taal met `if`/`while` voor de Java Virtual Machine. Daarnaast worden strings aan de compiler toegevoegd. De compiler is helaas zeer matig getest wat resulteert in een aftrek van 1.0 punt. Het cijfer zou dan zijn: 7.5 + 0.0 - 1.0 = 6.5.
- Een tweetal studenten implementeert een compiler voor de expressie-taal voor de Triangle Abstract Machine. Daarnaast worden enumerated types en records toegevoegd. De verslag en de tests zijn echter zeer goed, waardoor een punt extra wordt verdiend. Het cijfer zou dan zijn: 6.0 + 0.25 + 0.25 + 1.0 = 7.5.

De deadline voor het eindproduct van de eindopdracht Vertalerbouw is **woensdag 6 juli 2011 om 17.00 uur**. Deze deadline voor het inleveren van het verslag is **strict**. Op het te laat inleveren staat de volgende sanctie:

- *één werkdag* te laat: `-0.5` punt voor de eindopdracht;
- *twee werkdagen* te laat: `-1.0` punt voor de eindopdracht;
- *drie werkdagen* te laat: `-1.5` punten voor de eindopdracht;
- *vier werkdagen* te laat: `-2.0` punten voor de eindopdracht;
- *vijf werkdagen* te laat: `-3.0` punten voor de eindopdracht;
- *nog later*: maximaal een `4.0` voor de eindopdracht.

A.3 Verslageisen

Het eindproduct van de eindopdracht Vertalerbouw dient uiterlijk **woensdag 6 juli 2011 om 17.00 uur** te worden ingeleverd in de doos bij kamer Zilverling 5037 of in het postvakje van Michael Weber (kamer 5110) op vloer 5 van het Zilverling-gebouw (leerstoel FMT).

Geef op het titelblad van het verslag duidelijk aan wie de auteurs zijn van het verslag. Dit betekent voor elke student: *achternaam, voorletters, studentnummer en adres en de naam van de studentassistent (*plus *“herhaler” voor studenten die het vak dit jaar herhalen).*

Het eindproduct zelf bestaat uit twee delen:

- Een *CD-R* met de ontwikkelde programmatuur. Hieronder is in meer detail aangegeven wat zich *in ieder geval* op de CD-R dient te bevinden. Ook op de CD-R dient duidelijk vermeld te zijn wie de makers zijn.
- Een *uitgeprint verslag* over de eindopdracht. Hieronder is te lezen wat hier allemaal deel uit van moet maken.

Daarnaast dient elk verslag vergezeld te worden met, voor iedere student, (i) een ingevuld tentamenbriefje en (ii) een volledig ingevuld evaluatieformulier (beiden zijn beschikbaar bij de kamer van Michael Weber).

Inleveren per email is *niet* toegestaan. Wanneer het ingeleverde op wezenlijke punten afwijkt van het hier gevraagde komt het niet voor beoordeling in aanmerking.

A.3.1 Programmatuur

De *opgeleverde programmatuur* (dat wil zeggen, de ingeleverde CD-R) dient de volgende delen te bevatten:

- Een `README`-file met daarin aanwijzingen voor de installatie en het opstarten van de vertaler, zoals de voor executie noodzakelijke directories en files en de manier waarop e.e.a. geïnstalleerd en aangeroepen dient te worden. Bij het lezen van deze file moet een gebruiker in staat zijn de vertaler foutloos te genereren, te compileren en op te starten. *Indien hieraan niet voldaan is, komt het programma niet voor beoordeling in aanmerking; niet-compileerbare programma's worden niet geaccepteerd.*
- De volledige ANTLR specificaties en de Java files die door ANTLR gegenereerd zijn;
- De Java-code van alle zelfgedefinieerde klassen, in één enkele directory-hiërarchie. De code dient aan de volgende eisen te voldoen:
  - Foutvrij te compileren;*
  - Specificatie (in javadoc) van klassen en methoden;*
  - Zinvol gebruik van packages en toegankelijkheden;
  - Begrijpelijke opmaak en naamgeving, volgens Java-conventies;

  De met een sterretje (*) gemarkeerde eisen zijn noodzakelijk om voor de eindopdracht minstens een 4.0 te krijgen.
- Documentatie (door `javadoc` geproduceerd in `html`) van alle zelfgedefinieerde klassen, in een eigen directory-hiërarchie (dus niet gecombineerd met de Java-files).
- Bytecode van eventuele gebruikte voorgedefinieerde klassen, voor zover het geen zelfgeprogrammeerde of standaard Java-klassen betreft.
- Resultaten van alle uitgevoerde tests.
  Voor *correcte* testprogramma's komt dit per test neer op:
  - het correcte programma zelf,
  - de gegeneerde TAM-code, en
  - enkele testruns (invoer en uitvoer)

  Voor *incorrecte* testprogramma's komt dit per test neer op:
  - het foutieve programma zelf, en
  - de uitvoer gegenereerd door de compiler (d.w.z. de gegeneerde foutmeldingen)

Zorg ervoor dat u het systeem zodanig oplevert dat de beoordelaar na het lezen van de bijgeleverde `README` de vertaler kan genereren, compileren en executeren. *Het mag dus niet nodig zijn iets in de source-code te veranderen!* Typische gevallen waarin dit fout gaat zijn: namen en paden van files of andere URLs, zoals van hostmachines van servers. *Test dit alvorens uw product op te leveren.*

A.3.2 Verslag

Het verslag moet inzicht geven hoe de taal gedefinieerd is, en hoe de problemen die zich voordeden bij het maken van de vertaler opgelost zijn. Vermeld ook wie voor welk onderdeel verantwoordelijk is en welke delen samen gemaakt zijn. Het verslag van de practicumopdracht dient in ieder geval de volgende onderdelen te bevatten:

- *Inleiding.* Korte beschrijving van de practicumopdracht.
- *Beknopte beschrijving* van de programmeertaal (maximaal één A4-tje).
- *Problemen en oplossingen:* uitleg over de wijze waarop je de problemen die je bent tegengekomen bij het maken van de opdracht hebt opgelost (maximaal twee A4-tjes).
- *Syntax, context-beperkingen en semantiek* van de taal met waar nodig nadere uitleg over de betekenis. Geef de beschrijving bij voorkeur in dezelfde terminologie als die gebruikt is bij de beschrijving van Triangle in Watt & Brown (hoofdstuk 1 en appendix B).
- *Vertaalregels* voor de taal, d.w.z. de transformaties waaruit blijkt op welke wijze een opeenvolging van symbolen die voldoet aan een produktieregel wordt omgezet in een opeenvolging van TAM-instructies. Vertaalregels zijn de 'code templates' van hoofdstuk 7 van Watt & Brown.
- *Beschrijving van Java programmatuur.* Beknopte bespreking van de extra Java klassen die u gedefinieerd heeft voor uw compiler (b.v. symbol table management, type checking, code generatie, error handling, etc.). Geef ook aan welke informatie in de AST-nodes opgeslagen wordt.
- *Testplan en -resultaten.* Bespreking van de 'correctheids-tests' aan de hand van de criteria zoals deze zijn beschreven in het §A.5 van deze appendix. Aan de hand van deze criteria moet een verzameling test-programma's in het taal geschreven worden die de juiste werking van de vertaler en interpreter controleren. Tot deze test-set behoren behalve correcte programma's die de verschillende taalconstructies testen, ook programma's met syntactische, semantische en run-time fouten.
  Alle uitgevoerde tests moeten op de CD-R aanwezig zijn; van één testprogramma moet de uitvoer in de appendix opgenomen worden (zie onder).
- *Conclusies.*

In de *appendix* van het verslag dienen ten minste de volgende onderdelen aan het verslag te worden toegevoegd:

- ANTLR `Lexer` *specificatie.* Specificatie van de invoer voor de ANTLR *scanner generator*, d.w.z. de token-definities van het taaltje.
- ANTLR `Parser` *specificatie.* Specificatie van de invoer voor de *parser generator*, d.w.z. de structuur van de taal en de wijze waarop de AST gegenereerd wordt.[2]

[2] Net als bij `Calc.g` is het uiteraard toegestaan om de specificatie van de `Lexer` en `Parser` te combineren in een gezamenlijke `grammar` specificatie.

- *Alle* ANTLR `TreeParser` *specificaties.* Waarschijnlijk zult u (tenminste) twee tree parsers gebruiken: een context checker en een code generator.
- *Invoer- en uitvoer van één uitgebreid testprogramma.* Van één correct en uitgebreid testprogramma (met daarin alle features van uw programmeertaal) moet worden bijgevoegd: de listing van het oorspronkelijk programma, de listing van de gegenereerde TAM-code (bestandsnaam met extensie `.tam`) en één of meer executie voorbeelden met in- en uitvoer waaruit de juiste werking van de gegenereerde code blijkt.

Zorg ervoor dat de bovenstaande listings *goed leesbaar* zijn; d.w.z. in ieder geval geen linebreaks in de uitvoer bevatten. Het is toegestaan om de listings in *landscape*-oriëntatie af te drukken. Gebruik bij het printen dezelfde tab-stops als u in uw programma-editor gebruikt heeft.

De reden dat de listings van de programmatuur in het verslag moeten worden opgenomen (terwijl ze ook al op de CD-R staan) is tweeledig. Ten eerste geeft het inzicht hoe uw compiler is opgebouwd; het geeft aan hoe de syntax, contextbeperkingen en vertaalregels in ANTLR zijn uitgewerkt. Ten tweede vergemakkelijkt het de correctie van uw werk: het verslag vormt de primaire basis van de beoordeling; de programmatuur op de CD-R wordt alleen gebruikt om uw compiler te testen. Het moet voor de correctie van uw werk niet nodig zijn om uitgebreid de programmatuur op de CD-R te bestuderen; de appendices moeten voldoende inzicht geven in het ontwerp en opbouw van uw compiler.

A.4 Eindopdrachten

Deze sectie is gebaseerd op sectie "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.

The student is invited to develop a complete compiler for this language. Using the compiler generation tool ANTLR, the student has to build

- a *scanner* (= lexer), which translates a stream of characters to a stream of tokens;
- a LL(1) *parser*, which checks whether the syntax of the program is correct and builds a abstract syntax tree (AST) from the stream of tokens;
- a *context analyzer* (an ANTLR tree parser), which checks whether the scope- and type rules of language are obeyed; and
- a *code generator* (also an ANTLR tree parser), which generates object code from the AST.

While the student may find suggestions for the syntax of this language in this chapter, *the student is encouraged to choose his or her own syntax.* It is suggested that a lexical specification be developed first, followed by an extended context-free grammar.

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`

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

Tabel A.3: Arithmetic operators, their relative priorities, and the types of their operands and result.

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 extra scope rules. We then present some more complicated extensions: procedures, functions, pointers (absolute addresses), arrays, and records.

A.4.1 Basic expression language

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 and denotation, 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 A.3. 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);
     } ;
```

Note that in this example we used the curly braces '{' and '}' characters to enclose the compound expression into a closed compound expression. 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. [3]

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.

A.4.2 Conditional statement

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:

[3] 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.

- 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.
  - 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.

A.4.3 While statement

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*.

A.4.4 Procedures and functions

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.
```

A.4.5 Arrays

`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 A.3.

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.

A.4.6 Records

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 A.3.

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.

A.4.7 Pointers

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.

A.5 Testen

Deze sectie is gebaseerd op sectie "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].

To build a compiler, a language specification consisting of a scanner and parser part must first be developed. This specification must define the syntax of the language that is required. Tree walkers that perform context analysis and code generation have to be specified as well. The result of feeding these specifications to ANTLR is a compiler that translates a source program written in the given language into a target program. The compiler behaves correctly if every target program conforms to the language specification.

Note that language specifications are themselves not completely formal. The definition of Triangle (Watt & Brown, 2000, appendix B), for example, uses a *formal extended context-free grammar* to define the context-free syntax, but *informal natural-language sentences* to describe the context-sensitive syntax and the semantics. To formally verify the correctness of the Triangle compiler would require a completely formal specification of Triangle, which unfortunately does not exist.

Instead of verifying our compiler, we could 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 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 §A.4.1).[4] By studying this test set the reader will be able to build an extensive test set for his or her own language. In subsequent sections we will discuss the construction of test sets for each of the different extensions to the basic expression language.

A.5.1 Basic expression language

For the basic expression language of §A.4.1 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’;
```

[4] The syntax of the language as used in this section just an example. The student is free to choose its own terminals and symbols for his or her own language.

```
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.
```

Figuur A.1: Test program that checks for correct syntax.

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 A.3 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
```

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);
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 A.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 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 A.2. Note that Java-style line comments are used are used to specify comments (i.e., `//` . . .).

```
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.
```

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

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 context analyzer. 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 A.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 A.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.

A.5.2 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.

A.5.3 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,

```
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.
```

Figuur A.3: Test program that checks context constraints.

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

Figuur A.4: Test program that contains a run-time error.

requires special attention. Furthermore, the scope rules that apply to the boolean expression should be taken into account.

A.5.4 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.

A.5.5 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.

A.5.6 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.

A.5.7 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.