Preface

This report is part of a project on the subject of languages and compilers, on the sixth semester of Bachelor of Science Software Engineering.

The report is produced at Aalborg University Esbjerg in the period 20 march 2002 to 31 may 2002.

The report has been composed by Jeppe Moritz Hansen, Mikael Heinze, Lars Lodberg Hundebøl, Jens Iwan Jørgensen, Tommy Godsk Jørgensen and Claus Hallas Nielsen. Supervisor is Jesper Kjær Pedersen.

Contents

1. Abstract

The ambition of the following project was to define a language for LEGO Mindstorm and Cybermaster controller units, known as RCX-units. Afterwards a compiler capable of translating the defined language to code, that could be uploaded and executed in this environment would be implemented. During the course of the project it turned out that the RCX-unit was very limited in its capabilities and some changes had to be made in the code generation phase. If an important part of the compiler construction theory was not to be skipped. Unfortunately that meant that showing something actually working in the RCX environment was not a possibility.

In this document we will describe the implementation of the compiler. It is implemented in an object oriented framework generated by SableCC. The document will describe the various phases; grammar specification, error checking and handling, symbol table implementation and code generation. Further a description of problems, choices and theory, important to the project, will be documented. The final product consists of this report and a compiler capable of translating a limited Java like syntax language to a small assembler language for the RCX-unit.

1. Problem definition

In this project, we are to build a compiler which input, is of a language define by us, and translates this into code for use with an RCX-unit.

A compiler generation tool will be chosen from the broad variety of tools available, and we will select the most appropriate tool, by matching the capabilities of these tools. During the project we will show, by constructing and documenting a compiler, that we have understood the theory and practice of building a compiler.

We need to design a complete language and thereafter implement some or all of the phases in the compiler; lexical, semantic and syntactic error checks, along with the backend; code generation and optimization.

Limitations

The prime focus of this project is designing the language, building the compiler and the generation of the target code, we will not concern ourselves with the protocol and communication to the RCX-unit. Only limited number of commands for the RCX-unit will be implemented.

2. The RCX-unit

There are, as mentioned earlier two types of RCX-units, we will in this section go into further detail on the Cybermaster and Mindstorm units.

Facts about the units:

- CPU: Hitachi H8 microcontroller with 32K of external RAM
- 16K ROM contains the firmware and driver for the unit
- Programs are downloaded to the RCX as byte code
- 6K is the maximum capacity for user programs
- Both units communicate with a pc trough the serial port (2400 baud, 1 start, 8 data, odd parity, 1 stop bit). The Mindstorm unit trough an IR-connection[1] whereas the Cybermaster communicates via antennas (radio waves).

[1] Infra-red connection

- Mostly all commands are identical for both units, only a few are restricted to be used with the Mindstorm unit only.
- LEGO has decided to cancel the Cybermaster and only to market the Mindstorm unit, the new versions of the unit communicates via USB trough an IR-connection.

Both units have limitations on the format of data and the amount stored:

- 8 Subroutines per program
- 10 Tasks per program
- 32 Variables
- No function calls
- Only integer and boolean as data types
- Only the Mindstorm unit supports data collection and reporting

2.1. Programming the RCX-unit

A very used programming tool is NQC. Below is an example of syntax of NQC, the example program makes the RCX-unit drive forward for 5 seconds and stop.

```
task main()
{
  OnFwd(OUT_A+OUT_C); // Starts motor A and C makes them go "forward"
  Wait(500);          // Waits for 5 seconds
  Off(OUT_A+OUT_C);   // Turns of motor A and B
}
```

Example 1 – Simple NQC code

The RCX-unit interprets the downloaded program line by line. It first turns motor A and C on, then goes on to the next line, waits 5 seconds and finally turns motor A and C off.

Below is another NQC program that repeats a subroutine 10 times. The subroutine turns the RCX-unit left for a decreasing number of seconds. This example shows how global variables are used as input for subroutines. This technique can also be used to carry return values, as the RCX-unit does not offer this facility.

```
int seconds;              // Init - the variable seconds
task main()
{
   seconds = 10;          // Sets seconds to 10
   while (seconds >= 1)   // Repeats the statement
   {                      // until the condition is met
      OnFwd(OUT_A+OUT_C); // Starts motor A and C makes them go
                          // "forward"
      Wait(500);          // Waits for 5 seconds
      Off(OUT_A+OUT_C);   // Turns of motor A and B
      turnLeft();         // Calls the subroutine turnLeft()
      seconds--;          // Sets seconds = seconds -1
   }
}

sub turnLeft()          // Declaration of the subroutine turnLeft()
{
   OnFwd(OUT_A);          // Sets motor A forward and turns it on
   OnRev(OUT_B);          // Sets motor A reverse and turns it on
   Wait(seconds*100);     // Waits for x seconds
   Off(OUT_A+OUT_B);      // Turns of motor A and B
}
```

Example 2 – Simple NQC code with loop

Each variable, task and instruction and its parameters is translated to hex-values. The hex-values are then ready for upload to the RCX.

2.2. RCX functions

The language for the RCX-unit will be supported by a small set of functions. Although many more functions easily could be well defined and useful in being added, we have chosen to limit these to the motor operations, which are needed for any program to the RCX-unit.

In addition to this set of functions, we will also include reserved keywords for use with these. The constants are added for simplicity when writing a program in the language.

3. Paradigms

This section describes which paradigm category the developed language belongs to. First a brief description of the commonly known paradigms, then followed by a categorization of the language.

The primary paradigms are imperative procedural, object oriented, logic and functional. The imperative procedural and object oriented belongs to a group of imperative paradigms. Where as the logic and functional paradigms are found in the group of declarative paradigms. Each paradigm fits to a given domain of how the programming language works.

Logic programming is based on rules of inference and queries. A query will be solved on the background of a set of facts and rules. Logic programming is good for solving problems about proofs.

Functional programming is inspired of the principle of functions in mathematics. A function is called with an input and then delivers an output related to the input.

Object oriented programming is based on modelling. Data and operations are encapsulated in different parts of the model called a class. Only a part of the set of data and operations in a class are accessible for the rest of the classes. This is how the classes are exchanging data.

Procedural programming has focus on the task that is to be executed. It controls the single steps of the task. Sometimes procedural programming is referred to as a higher abstraction of assembly language.

Related to the brief description of the paradigms, each of the paradigms will be taken into consideration. The purpose of the language is to make it possible to control the RCX-unit connected to the computer. You have to define the task that the RCX-unit has to perform. When the focus here is on the task to be performed, it will be natural to choose the procedural paradigm.

An object oriented paradigm could come into consideration. You could imagine that the RCX-unit has to interact in an environment, where other active components, will have influence on its tasks. Therefore an exchanging of data should be possible. The fact is that the RCX-unit will not interact with other active components. A practical

reason is that it would make the project too difficult and have a range bigger than the defined problem.

A functional paradigm will not be useful in this case, because it is the ability to perform certain tasks and not to generate output based on a given input, which has our focus. On the other hand it could be smart to have the principle of functions in the language. With the procedural paradigm, some procedures are defined. These procedures group a set of connected steps of the task. It could then be nice to have the ability of returning a result after ending the execution of a given procedure. This adds the property of a function.

The last paradigm, the logic, will not be an appropriate choice, because the concept is that the control flow in a program is based on a set of predefined facts, which results in a yes or no answer.

In the RCX-unit, it is tasks that need to be executed and not queries that need answers. As it seems the choice will be the procedural paradigm, with the property of defining some of the procedures as functions with a returning value. Compared to the target code, which is an assembly like language, this will be another argument to the choice of the procedural paradigm.

4. The choice of a compiler compiler

Upon building a compiler, one of the first choices is to select the appropriate tool. We wish to make use of a compiler tool, as making a compiler by hand is a cumbersome task[2].

Several compiler development tools have been created. These tools are also better known as compiler compilers. A compiler compiler relieves the programmer from the burden of writing the lexical and syntactical analysis code. The compiler compiler uses the grammar specification, written in Backus-Naur Form[3] (BNF), of the language to generate the lexical and syntactical analyzer.

Since we have no experience with any compiler compiler, our approach will be to first examine some popular and widely-used compiler compilers and see how these operates, and compare them.

The following tools went in to our consideration of choosing the best compiler compiler for this project.

Flex/Bison is a popular version of the older Lex/Yacc compiler compilers controlled by the Open Software Foundation GNU. They are a pair of tools that together can generate a compiler or its front-end using the C language to specify the action code[4].

JFlex/Cup is Flex/Bison ported to the Java environment.

JavaCC is a reaction to the complexity of using tools like Flex/Bison. JavaCC has combined the functionality of the lexer and parser into one tool which is easier to use.

[2] It took 18 staff-years building FORTRAN in the 1950's, which were made from scratch.
[3] BNF also known as context free grammars, is a notational tool for describing the syntax of a language.
[4] Action code are sometimes also referred to as user code (code written by the programmer)

In combination with a tool called JTree it is possible to build an Abstract Syntax Tree otherwise known as AST[5].

SableCC differs from the other compiler compilers by using an object oriented framework. This means that an action is added by defining new classes containing the action code. Like JavaCC SableCC uses AST for constructing parse trees, but unlike JavaCC, the integrity of the AST is not left in the hands of the programmer.

A common denominator for Flex/Bison, JFlex/Cup and SableCC is that they all use bottom-up parsing. JavaCC is the exception and uses top-down parsing.

In the following we have tried to contrast the differences between each compiler compiler.

	Flex/Bison	JFlex/Cup	JavaCC	SableCC
Programming Environment	C	Java	Java	Java
Grammar	BNF	BNF	EBNF[6]	EBNF
Parsing	Bottom-up	Bottom-up	Top-down	Bottom-up
Object Oriented Framework	No	No	No	Yes
Source code availability	Yes	Yes	No	Yes

Figure 1 - Compiler compiler comparison chart.

4.1. Selecting a compiler compiler

After experimenting with each of the above tools, we concluded the following.
As we are not very experienced with building a compiler, we needed a tool that was easy to learn and easy to use. Because of our familiarity with the Java programming environment, we reduced the decision to be either JFlex/CUP, JavaCC or SableCC.

JFlex/CUP was eliminated from the decision, since the AST has to be created manually, and the interconnectivity between the two tools is not very maintainable.

SableCC is a bottom-up compiler. This means that we do not need to concern us with problems with handling left recursion (this subject is discussed in chapter 8.4). On the other hand JavaCC is a top-down compiler. Here it is important to eliminate left recursion; otherwise it is possible for the parser to loop forever.

After studying both JavaCC and SableCC closely, we decided to use SableCC because of its object oriented framework which made it very easy to extend the compiler, as we saw fit (more on this subject later). Furthermore we did not have to concern us with handling left recursion.
SableCC also had a much easier way for building parse trees than JavaCC.

[5] Abstract Syntax Trees are used to develop parsers, programs that interpret the meaning of expressions written in a concrete syntax. That is why Abstract Syntax Trees are often just referred to as parse trees.
[6] Extended BNF is BNF augmented with the regular expression operators: (), *, ? and +

To help us with error recovery we are to use SableCC version 2.17.3 experimental instead of the more stable SableCC 2.16.2 final, which did not have error recovery features. No problems with the experimental version were encountered.

5. The SableCC framework

The idea behind SableCC and its object oriented framework is that it should be more maintainable than other current compiler compilers on the market. The extensive use of object oriented design patterns were made to achieve modularity of code.

Also, when designed, SableCC was to meet the new compiler implementation trends. In other words a compiler which could implement multiple passes over the compiled program, the use of AST and the possibility to extend the compiler in the near future with added analyses and optimizations.

We will in the following describe each step for building a compiler (or interpreter) using SableCC. These steps are illustrated in Figure 2 .

1. First the SableCC grammar specification file is created describing the lexical definition and the grammar.
2. The grammar is then used to generate the framework.
3. One or more working classes[7], created by the programmer, can then inherit from this framework.
4. Together with the main compiler class, this activates first the lexical analyzer and then the parser together with any working classes that may be needed.
5. The compiler is then being compiled using a regular Java compiler.

[7] Working classes is classes that contain the core compiler functionality, like analysis and code generating.

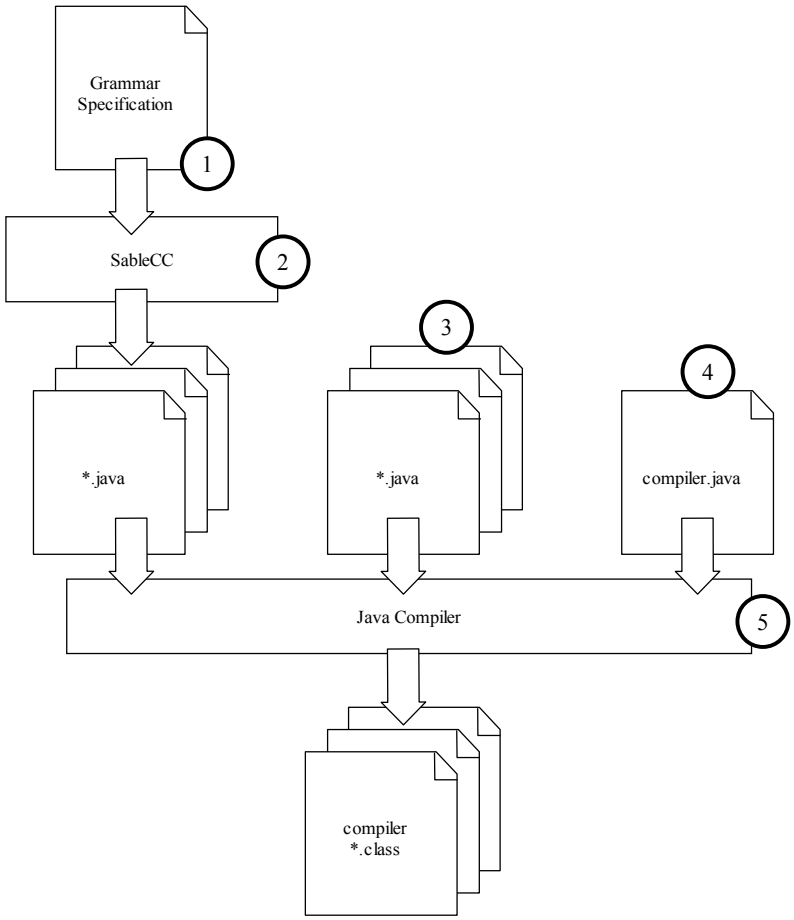


Figure 2 - The steps for creating a compiler in SableCC

The SableCC grammar specification file is simply a text file containing both the lexical definition and the grammar productions. It is not possible to add action code to the specification file.

All action code must be added in classes inheriting from the working classes. This is also why adding, changing or removal of action code does not affect the grammar.

When generating the framework, SableCC divides the generated files into different subdirectories. The **lexer** subdirectory contains the generated lexer and the lexer exception classes, in case an error occurs in the lexical analysis phase. This is also the case in the **parser** subdirectory, where the parser and the exception handler generated files are located. The **node** subdirectory is where all the classes for defining the AST are located. Classes that represent tokens are named with a capital T following by the token name. Classes that represent productions are named with a capital P following by the production name. A production can have multiple alternatives these are prefixed with an A, and suffixed with the name of the production. SableCC requires that alternatives are named, this is done by prefixing the alternative with a name inside braces, as shown below.

```
production =
    {alternative_1} <terminals> and <non-terminals>
    | {alternative_2} <terminals> and <non-terminals>
    ;
```

Finally the **analysis** subdirectory contains classes for defining the AST walkers.

Furthermore we use version 2.17.3 experimental of SableCC which has an error recovery facility. This results in a generated **error** subdirectory, where error and error exception handler packages resides. Each of these subdirectories is represented as packages.

5.1. The SableCC tree walker

To traverse the AST is by using the so called tree-walkers. In SableCC the AST walkers is represented as a class that will visit all the nodes in a predefined order. It is possible when using the tree walkers in SableCC to visit all the nodes in a depth-first traversal or in a reverse depth-first traversal.

The SableCC AST walker classes are extended to make it possible to add actions on specific nodes, and the way this is done, is by using an adaptation of an extended visitor design pattern[8].

To make it impossible to extend a node class to store information directly in the node, all AST node classes are declared final. Information about each node can instead be obtained by the AnalysisAdaptor class (a part of the analysis subdirectory). The AnalysisAdaptor class has methods for saving and retrieving information to and from internal hash tables.

6. Informal language definition

The following paragraph will be an informal description of our programming language. The informal description here reflects the language after we have changed the target from RCX understandable code, to assembler code. The text is not in any way an in depth description of our syntax. Its intention is to give the developers an early and joint understanding of our language in the development process. Finally it should give the reader of this documentation an introduction to the syntax.

From this point on in the report, we use the word procedure as a generalization of procedures, functions and routines.

- Each program is made up of global identifiers and procedures. First part of the program is the global identifiers, second part is the procedures.
- Identifiers can be either variables or constants. They can be declared either as local or global identifiers.
- Two data types are supported by the language. Boolean which can have the values true or false and integers which value can range from -32768 to 32767[9].
- Identifier assignment can be done by expressions. (e.g. int a = b + c or boolean a = b < c)
- Procedures can have parameters and return values. A procedure with a return value is normally referred to as a function.
- Parameters in procedures are called by value.
- Each program must have exactly one procedure named main. This is where program execution starts.
- Conditionals is made by if statements. If statements can be followed by one else block. ‘Else if’ blocks is not supported.

[8] The extended visitor design pattern makes it possible to add a new class without modifying any existing classes by using an overall interface. More specific SableCC implements a utility class called AnalysisAdaptor that implements Analysis and provides a default implementation for all methods.
[9] In many languages this range is considered short, we will however refer to them as integers.

- Loops is made by the while or repeat keyword. A while loop will run as long the condition is evaluated true or a break statement is reached. A repeat loop will run the amount of times given in its expression.
- Scopes are created with the following lexemes '{' and '}'.
- The language is case sensitive.
- Semicolon is used as statement terminator.
- Comments in the source text can be written in two ways. Either as a block which can be started with an '/*' and must be terminated with an '*/' or as a single line comment started with a '//' and terminated at end of line.
- RCX commands are reserved keywords.

7. An introduction to the compiler

This chapter will explain the concepts behind the compiler.

The compiler is based on the analysis synthesis model of compilation. The analysis part breaks up the source program into constituent pieces and makes an analysis of these. The synthesis part generates the program into the target language which in our case is the RCX assembler code. More specifically the analysis synthesis model is broken up into phases. Each of these phases will be described in details in theory and in practice.

7.1. The phases of the compiler

The compiler consists of four different phases. The analysis part consists of a lexical analyser, a syntax analyser and a semantic analyser. Theses phases all use error handling procedures when an error is encountered. This part of the compiler represents the front end.
At the back end resides the code generator. The code generator and the semantic analyser use the symbol table.

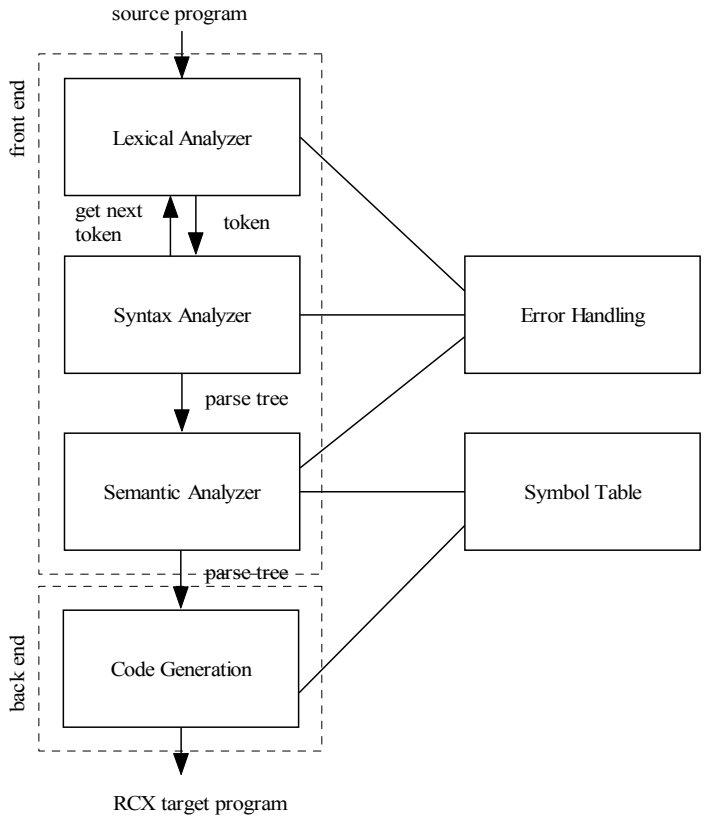


Figure 3 - Phases of the compiler

Other commonly known phases of a compiler are the intermediate code generator and the code optimizer. Both these phases were considered but not included into the compiler.

The intermediate code generator translates the source program into some kind of intermediate code which is then again translated into the final target language. The decision for disregarding this phase was chosen because there were no benefits for doing so. A benefit for using intermediate code is if the target machine should change. In our project this is not very likely[10].

In general the code optimizer uses the intermediate code to optimize upon. Code optimization is used to improve the intermediate code so, when translated to the target code it will require less memory and run faster.
The code optimization phase can also be implemented as part of the code generation phase. This is also referred to as peephole optimization and is a way to optimize directly on the target code.

7.2. Compiler executions

The diagram above illustrates how the compiler structure is organized. In this section the flow of execution will be described.
The compiler compiles the source program into the target program in four steps.

1. The lexical analyser reads the source program from start to finish and converts the input into a stream of tokens which is then used to generate the syntax

[10] Although the RCX-unit consist of two different types, the Mindstorm and the Cybermaster units, the RCX assembler language is mostly the same, except for some minor restrains to the Cybermaster unit.

tree. Errors during this process will here be thrown if any exists. White space[11] and comments is ignored.
2. First traversal of the syntax tree fills the symbol table.
3. At the second pass through the syntax tree the error productions and semantics are checked at the same time.
4. The third and final traversal of the syntax tree generates the target code. If errors occur during semantic and/or error production checks then this phase of the compiler is not executed.

8. Source language

The following section will describe our grammar file. This grammar file will define the syntax of our compiler. From this grammar file, SableCC will generate a lexical analyzer and the parser. It should be noted that SableCC takes one file with both the lexer and parser specification. The lexer generator and parser generator is not two separate programs. The language we decided to define is heavily inspired by the java language, although some significant differences exist. An additional note is that the described syntax here does not include error productions.

8.1. Lexer specification

The lexer specification exists of three blocks. Helpers, Tokens and Ignored Tokens. Helpers are only used to help you build regular expressions, defined in the 'Tokens' block. Ignored tokens are tokens the lexer will skip when the parser is requesting next token. The complete lexer specification can be seen below.

```
Helpers
    all             = [0..255];
    digit           = ['0'..'9'];
    non_digit       = ['_' + [['a'..'z'] + ['A'..'Z']]];
    tab             = 9;
    cr              = 13;
    lf              = 10;
    space           = 32;
    eol             = ( cr lf | cr | lf );
    not_star        = [all - '*'];
    not_starslash   = [not_star - '/'];

Tokens
    comma           = ',';
    semicolon       = ';';
    l_par           = '(';
    r_par           = ')';
    l_brace         = '{';
    r_brace         = '}';
    mul             = '*';
    div             = '/';
    plus            = '+';
    minus           = '-';
    mod             = '%';
    not             = '!';
    assign          = '=';
    equal           = '==';
    noteq           = '!=';
    lt              = '<';
    gt              = '>';
    lteq            = '<=';
```

[11] Blanks, tabs and newlines.

```
    gteq            = '>=';
    and             = '&&';
    or              = '||';

    bool            = ( 'true' | 'false' );
    functions       = 'functions';
    if              = 'if';
    else            = 'else';
    while           = 'while';
    final           = 'final';
    void            = 'void';
    int             = 'int';
    boolean         = 'boolean';
    break           = 'break';
    repeat          = 'repeat';
    return          = 'return';

    gofwd           = 'goFwd';
    gorev           = 'goRev';
    setspeed        = 'setSpeed';
    stop            = 'stop';
    wait            = ‘wait';
    playtone        = 'playTone';
    left            = 'LEFT';
    right           = 'RIGHT';
    both            = 'BOTH';

    number          = ( digit )+;
    identifier      = non_digit ( digit | non_digit )*;

    blank           = ( space | tab | cr | lf )+;
    line_comment    = '//' [all - [cr + lf]]* eol;
    block_comment   = '/*' ( not_star )*  ( '*' )+  (  not_starslash  not_star*  '*'+  )*  '/';


Ignored Tokens
    blank,
    line_comment,
    block_comment;
```

The Tokens can be separated in to four sections. First the delimiters and operators are specified, second the keywords in the language. Next the RCX-unit commands are specified. Finally we specify lexemes that only can be identified by regular expressions. A few things should be commented in the specification.

1. It can be seen from the *all* helper, the language is designed to support the extended ASCII table. No real argument can be given for this decision, since the standard ASCII table supporting 128 characters should suffice. However considering the easiness of this implementation, there was no reason not to support this.

2. The *non_digit* helper is defining the non digit characters allowed in identifiers. Non digits are simply all characters from ‘a’ to ‘z’ (capital and non capital) and the underscore character. More characters could have been included, however that would have been a trivial task.

3. The *eol* helper is defining different end of lines on different operating systems. Since our compiler is written in Java, this was a necessary but luckily an easy thing to support.

8.2. Parser specification

The syntax specification is written in Backus-Naur form. However the following operators from the extended Backus-Naur form are supported on elements: '*', '+' and '?'. They carry the same semantic as in regular expressions. However these operators can not be used on grouped terminals or non-terminals in parentheses. Below is our parser specification. **Bold** are terminals *italic* are non-terminals.

Productions

program →
    global_identifier_block **functions** *function_declaration_block*
    ;

global_identifier_block →
    *global_identifier_declaration**
    ;

global_identifier_declaration →
    identifier_declaration **semicolon**
    ;

identifier_declaration →
    const_declaration
    | *var_declaration*
    ;

const_declaration →
    final *data_type* **identifier** **assign** *type_value*
    ;

var_declaration →
    data_type **identifier** **assign** *type_value*
    | *data_type* **identifier**
    ;

data_type →
    int
    | **boolean**
    ;

function_declaration_block →
    *function_declaration**
    ;

function_declaration →
    function_return_value **identifier** **l_par** *declarator_parameters*? **r_par** *function_block*
    ;

function_return_value →
    data_type
    | **void**
    ;

declarator_parameters →
    data_type **identifier** *more_declarator_parameters*?
    ;

more_declarator_parameters →
    comma *declarator_parameters*
    ;

function_call →
identifier l_par *function_call_parameters*? **r_par**
;

function_call_parameters →
function_call_value more_function_call_parameters?
;

more_function_call_parameters →
comma *function_call_parameters*
;

function_call_value →
type_value;

block →
l_brace *statement_list** **r_brace**
;

function_block →
l_brace *statement_list** **r_brace**
| **l_brace** *statement_list** **return** *expression* **semicolon r_brace**
;

statement_list →
block
| *while_loop*
| *repeat_loop*
| *if_selection*
| *statement* **semicolon**
;

statement →
function_call
| *assignment*
| **break**
| *rcxcommand*
| *identifier_declaration*
;

rcxcommand →
gofwd l_par *engine_selector* **r_par**
| **gorev l_par** *engine_selector* **r_par**
| **setspeed l_par** *engine_selector* **comma** *value* **r_par**
| **stop l_par** engine_selector **r_par**
| **wait l_par** *value* **r_par**
| **playtone l_par** *value* **comma** *value* **r_par**
;

engine_selector →
left
| **right**
| **both**
;

while_loop →
while l_par *condition* **r_par** *block*
;

repeat_loop →
repeat l_par *expression* **r_par** *block*
;

if_selection →
if l_par *condition* **r_par** *block else_selection*?
;

else_selection →
else *block*
;

assignment →
identifier assign *type_value*
;

type_value →
condition
;

condition →
condition **and** *equality*
| *condition* **or** *equality*
| *equality*
;

equality →
equality **equal** *relational*
| *equality* **noteq** *relational*
| *relationel*
;

relationel →
relationel **lt** *expression*
| *relationel* **lteq** *expression*
| *relationel* **gt** *expression*
| *relationel* **gteq** *expression*
| *expression*
;

expression →
expression **plus** *term*
| *expression* **minus** *term*
| *term*
;

term →
term **mul** *not_operation*
| *term* **div** *not_operation*
| *term* **mod** *not_operation*
| *not_operation*
;

not_operation →
not *factor*
| *factor*
;

factor →
l_par condition **r_par**
| *value*
| **bool**
;

value →
identifier

| **number**
| *function_call*
;

A few things should be commented about the grammar.

1. The first production *program* is the root of the grammar.

2. In production *repeat_loop,* specification for a repeat loop exists. The loop will simply run the amount of times specified. Such a simple loop was added although it does not exist in Java.

3. Looking at the productions: *while_loop*, *repeat_loop*, *if_selection* and *else_selection*. It can be seen that while, repeat, if or else statements must be followed by braces. This was done to make the language more logically and easily read for inexperienced programmers, and to eliminate any possibilities for dangling else, since braces will eliminate any ambiguity.

4. An issue with the grammar, discovered during the later stages of the compiler implementation, can be seen in the production *function_block*. It can be seen that the return value from a procedure must be placed as the last statement, and may only be present once. That we did not discover this right away was lack of attention. The result of this is that the following code is illegal.

```
if (x > 0) {
  return x;
}
else {
  return -x;
}
```

   This could have been eliminated by treating **return** as any other statement, and letting the semantic check catch any error this might allow. However when the flaw was discovered there was no time to change it. Since it would cause some major changes in the semantic check.

5. In the first production *program*, the keyword **functions***,* is used to separate global identifiers, from procedure declarations. This keyword was introduced to solve a reduce/reduce conflict in the parser. See the following chapter.

8.3. Left factoring

This is a technique to make grammar transformations for predictive parsing. Whenever we have two or more alternative productions, which expand the same non-terminal, we postpone the decision until we know which alternative we are encountering.

An example for this would be our problem for recognizing whether we were dealing with a globally assigned variable declaration or procedure declaration.

program →
    global_identifier_block
    ;

global_identifier_block →
    *global_identifier_declaration**
    ;

```
global_identifier_declaration →
                identifier_declaration
                ;

identifier_declaration →
        const_declaration semicolon
        | var_declaration semicolon
        | function_declaration
        ;

var_declaration →
        data_type identifier assign type_value
        | data_type identifier
        ;

function_declaration →
        data_type identifier l_par declarator_parameters? r_par function_block
        ;
```

Figure 4 – Part of the grammar which resulted in a reduce/reduce conflict.

The parser would in the *identifier_declaration* production not be able to determine whether it was dealing with a variable declaration or a procedure declaration, because SableCC has only one lookahead, and since the next terminal is **identifier** in both, it would need an extra lookahead, to tell if the next token would be an **assign** terminal or an **l_par** terminal.

Using the idea of the algorithm for left factoring a grammar[12], the extract of the grammar shown above, would be transformed into the following. Showing how the **functions** keyword could have been avoided.

```
program →
        global_declaration*
        ;

global_declaration →
        declaration semicolon
        | declaration assign value semicolon
        | declaration l_par declarator_parameters? r_par block
        | final declaration assign value semicolon
        ;

block →
        l_brace statement_list* r_brace
        ;

declaration →
        data_type identifier
        ;

data_type →
        int
        | boolean
        | void
        ;
```

Figure 5 – Left factored grammar.

[12] [ASU88], Algorithm 4.2, pp. 178

With this revision of the productions, in *global_declaration* there is only need for one lookahead (*declaration*) before the parser can decide which alternative of the production it is encountering.

The way the problem was solved in this project however was by introducing the keyword **functions** which would then be required, before any procedures could be declared.

8.4. Left recursion

Productions are often made up of terminals and other productions; sometimes a production is also made up of itself, meaning that it has recursion.

Having a top/down parser (otherwise known as a recursive descent parser), a recursion can result in the parser going in to an infinite loop.

The following left-recursive production is in the grammar, and had it been a top/down parser, a problem would arise.

expression →
    expression **plus** *term*
    | *expression* **minus** *term*
    | *term*
    ;

The problem is that when the parser applies this production, it never matches any terminal on the right side of the production because it keeps calling the leftmost production (expression itself) recursively.

A solution to solve this recursive problem is to introduce the epsilon[13] terminal to the grammar and re-arrange the production, so it will not go into an infinite loop. With the problem above the solution would be to move the recursion to the right by having a terminal on the left instead. This requires and extra production, but it will let the parser avoid the problem.

expression →
    term rest
    ;

rest →
    plus *expression*
    | **minus** *expression*
    | **epsilon** *//* empty token
    ;

Now the grammar has been made right-recursive, because the recursion is done rightmost on the right hand side of the production, thereby letting the parser consume the terminals as they appear.

For SableCC left recursion is not a problem, since it is bottom/up, and produces rightmost derivations, avoiding a never-ending left recursion.

[13] Epsilon – also denoted by the symbol ε, it is a special terminal which is empty.

8.5. Ambiguity

Grammars can some times become ambiguous, when more than one parse tree can be derived from the same string. The most common occurrences of this, is the precedence of operators and the dangling-else problem.

In the case of precedence the problem is how to interpret a string with two or more operators of different precedence order. The textbook [ASU88] example is the expression 9+5*2. In the grammar the precedence problem is solved by extending the grammar with the *factor* production, by moving the operators with the highest precedence deepest into the syntax tree.

expression →
    expression **plus** *term*
    | *expression* **minus** *term*
    | *term*
    ;

term →
    term **mul** *not_operation*
    | *term* **div** *not_operation*
    | *term* **mod** *not_operation*
    | *not_operation*
    ;

not_operation →
    not *factor*
    | *factor*
    ;

factor →
    l_par condition **r_par**
    | *value*
    | **bool**
    ;

The dangling-else problem is another textbook example. The problem exists in the form that whenever an *if*-statement contains another *if*-statement and then followed by an *else*-statement. A problem arises in the form of which *if*-statement does the else belong to, thereby the name “dangling-else”.

In this project it is not very relevant for the grammar, since conditional-statements (if else) are required to have a block encapsulating the scope of each part. However the most common solution is to associate the *else*-statement with the closest *if*-statement.

8.6. Scope

Along with the definition of the language, we will also specify the type of scope the language (our compiler) will use. The scope defines which non-local variables are available to the different scopes within the program.

A global scope is the scope, which all other scopes can refer to. Procedures have their own scopes and within these, blocks (such as while loops, if statements and etc.) can exist with their own scopes.

This is a rather simple concept, but it is important to define when a scope can use a non-local variable and when it can not. Because the programmer have to know what is allowed by the compiler, to write correct code.

Scope can be organized in three ways:

1. Lexical scope without nested procedures
2. Lexical scope with nested procedures
3. Dynamic scope

Common for the three scopes are that they can only access non-local variables in scopes higher than them selves.

Ad 1)
This is the type of scope that is also used by Java. It is called a block-structured language, and is given by the *most closely nested*[14] rule. The rule defines a block scope as all the variables within the block itself and any non-local variable, most be declared within another block, which includes the block.

Ad 2)
This type of scope has the same properties as the one above. But in addition it also allows for nested procedures, but like the lexical scope without nested procedures it follows the *most closely nested* rule, which means that two procedures nested within the same procedure do not share scopes, even if one of them calls the other. This is illustrated in Figure 6, where it shows where the variable “va” is referenced from (marked by the arrows, pointing to the origin).

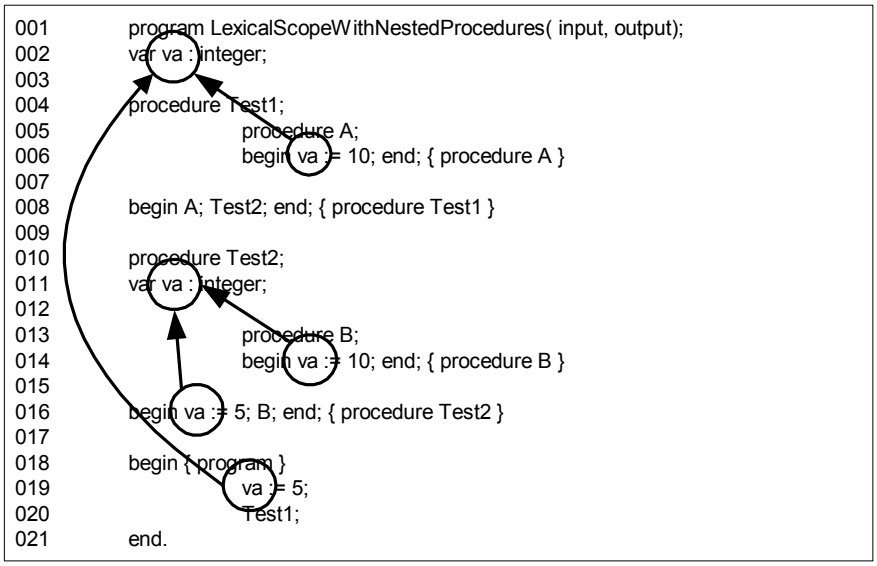


Figure 6 - Pascal program with nested procedures

Ad 3)
The dynamic scope differs from the lexical scope such that the scope of a procedure follows the called procedure. This means that if a procedure with a local variable “a” calls another procedure, then the variable “a” is available for this procedure as well.

The difference between lexical and dynamic scope is further illustrated by the following code example in Figure 7.

[14] [ASU88]: pp. 412

To show the difference between lexical (without nested procedures) and dynamic scope, we have provided a small Java program to illustrate the difference between the two scopes, and again it is shown from where the variable “va” is referenced. Note that the program-code is same, no matter the scope type, but the program do produce different output dependent of which scope is used.

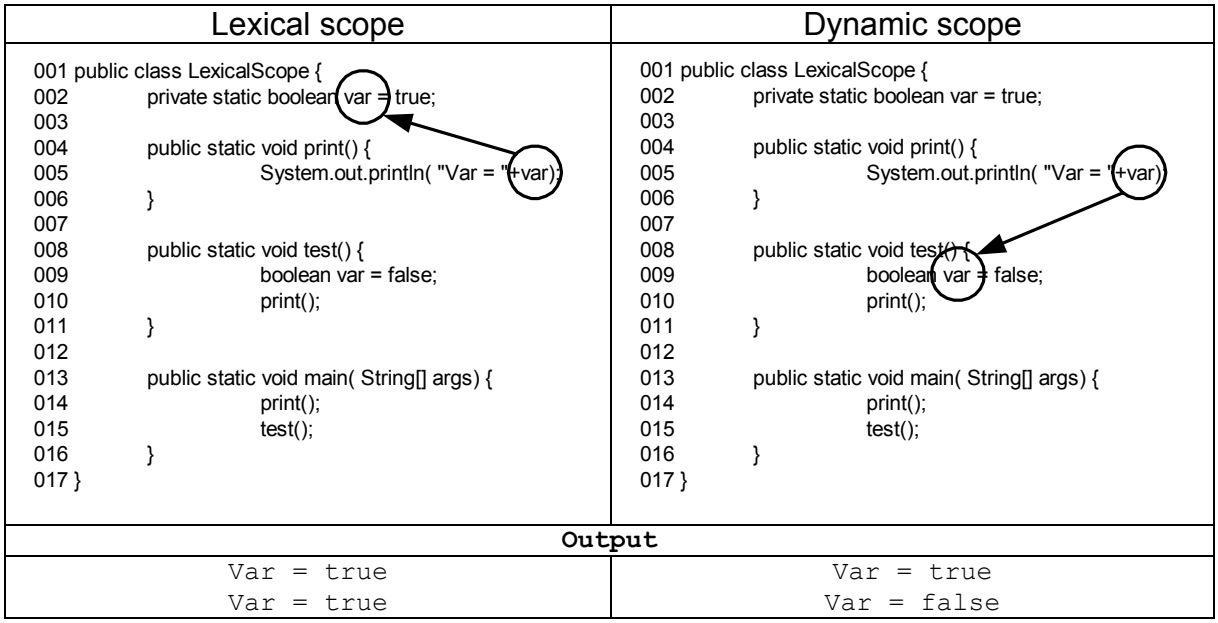


Figure 7 – Difference between scopes

Choice of scope type
Selecting a scope type for the language is created with the intention that the language should correspond as closely to the Java syntax as well as the type of scope. Java’s type of scope is the lexical scope without nested procedures.

We designed the grammar, so it would fit the lexical scope without nested procedures and thereby not supporting nested procedures. Where as the choice of selecting between the lexical scope without nested procedures and dynamic scope, would not be something that could be handled by the grammar, but would have to be done, during semantic checking and code generation.

We also had a personal preference for the lexical scope rather than the dynamic scope, which were that we found it simpler for the programmer to write a more readable/understandable code, when the scope is lexical.

The dynamic scope has the disadvantage (from our point of view), that when having an error somewhere in the code, it will be very difficult to debug. The problem is that the ability to call a procedure using another procedures locally variables may be fine. What if more procedures are calling the same procedure, will it then be able to perform correctly with the other or maybe it uses some other locally variables. Such code is very hard to maintain, without making sure that all procedures called by a procedure is updated as well.

9. Error recovery
During compilation of a program, a lot of errors might exist in the source program. It is desirable for a compiler to be able to report the presence and kind of an error,

recover from it, and continue checking the source. Different kind of errors might exist in a program

- Lexical: Such as illegal operators, keywords or other kind of tokens that are not supported by the language.
- Syntactic: Such as missing, too many, or misplaced tokens in statements. (e.g. missing parentheses or semicolons)
- Semantic: Such as values assigned to an incompatible variable.
- Logical: Such as infinite loops.

Semantic and logical errors can not be detected during parse time and must be handled elsewhere. Lexical and syntactic errors however are detected during parse time, which mean the lexer and parser in some way must be able to handle such an error.

9.1. Error recovery strategy

There exists different ways to handle errors during parse time. Some of the methods are:

- **Panic mode**: On an error the parser discards a part of the input. Usually a part of the program between delimiters such as semicolons or begin / end is discarded. This strategy is pretty simple to implement, however a large amount of the program might be skipped and not checked for further errors. In addition further errors might rise due to the fact that the correctness of some code might depend on the skipped block.
- **Phrase level**: If an error is detected the parser might try to remove, insert or replace some of the input in such a way that the parser might continue. Changes to the program can only happened at the current position of the lexer/parser. The choice of the correction is in the hands of the compiler designer. This method is pretty complicated to implement since the designer has to carefully choose a correct “fix” that does not need another “fix” and cause an infinite loop.
- **Error productions**: If we have a good idea of some common errors, we can design our grammar specification file (e.g. parser) in such a way that these common errors will still be accepted. We can then later on raise an error if any of the error productions has been created.
- **Global correction**: This is a method that will try to make as few adjustments in the incorrect input as possible. On an incorrect input an algorithm will try to make as few fixes globally in the input in such a way parsing can continue. This method is however to costly concerning time and space, which is why this technique is only of theoretical interest.

9.2. Chosen error recovery strategy

In the compiler we decided to combine panic mode and error productions. Phrase level and global correction was never seriously considered due to the fact that they would be to hard to implement.

The implemented panic mode, will on an error un-process tokens until it reaches a semicolon, a left brace or top of the file. When it has gone back to this previously error free state, we start to scan forward in the input, until we find a semicolon or a right brace. At this point parsing is resumed.

While scanning forward during error recovery none of the input tokens are processed. All input between are instead discarded. Depending on the errors found in the source, this strategy can give very mixed results. If an error is found in a simple one line statement (e.g. assignment or expression) and the error handler is able to recover without breaking any scope, this strategy works perfect.

However is the error found in a while loop, if condition, procedure header or anything else depending on subsequent scopes, a lot of errors might begin to arise in otherwise error free code. Another problem might be that a following error in the discarded scope is not found.

9.3. Error recovery in SableCC

Whenever an error is detected in a source file, the parser by default throws an exception with an error message containing the line number, what was expected and what was actually found. This exception can then be caught and printed to the screen. When such an error has been thrown from the parser there is no way to recover and continue parsing. However SableCC does offer an option to implement your own strategy to recover from a parse error. The SableCC auto generated parser class offer the following method:

```
public void setErrorHandler( ErrorHandler errorHandler );
```

You then make your own error handler class which implements the ErrorHandler interface. The parser will then use your error handler, instead of the default error handler. The ErrorHandler interface forces you to implement the following methods:

```
void handleLexerError( ErrorRecoverer p_recoverer, LexerException pe);
void handleParserError( ErrorRecoverer p_recoverer, ParserException pe);
```

If the parser detects an error, an exception will be thrown through the parser and be caught in the following method. This is the method the user of the parse class calls to start parsing the source. (The return object *start* is the root node of the generated AST)

```
public Start parse() throws ParserException, LexerException, IOException
{
   try {
        // Code to parse a source file.
        // Exception will may be thrown to this point.
   }
   catch ( LexerException e ) {
        errorHandler.handleLexerError( errorRecoverer, e );
   }
   catch ( ParserException e ) {
        errorHandler.handleParserError( errorRecoverer, e );
   }
   return …; // Returning the tree
}
```

Example 3 – The parse method of the Parser

The default error handler would simply throw the exception back out and the parsing would at this point stop. However the error recoverer which is given to the error handler along with the exception offers the following methods to help recover from parser and lexical errors:

`void processToken( Token p_token );`	Makes the parser process tokens (as if they were coming from the lexer). This lets one put the parser in any state before continuing the parsing.
`Token unprocessLastToken();`	Makes the parser 'go in reverse'. It tells it to go back to the exact state it was in before it encountered the previous token. It can be repeated multiple times to go back to any point.
`int readChar();`	Reads forward in the lexer input stream one character at the time.
`int readCharBackwards();`	Reads backwards in the lexer input stream one character at the time.
`int getLexerState();`	Returns the state of the lexer.
`void setLexerState( int p_state );`	Sets the lexer in a specified state.

With these methods manipulating the input and recovering from an error is fairly simple.

10. Error productions

Most common errors are due to programmers leaving out delimiters or simply using syntax from other similar programming languages, an example of this could be the use of ‘=‘ instead of ‘==‘ when comparing variables.

Handling these errors separately makes it possible to return exact error messages to the programmer and thereby shortening the debugging time. The way this is done is by defining productions for the most common errors. This makes it possible to catch the making of those productions when parsing the source code and let the parser continue, even if there are syntactical errors.

Making error productions is like allowing errors in the grammar, but when the syntax-tree is built and then later traversed, reaching an “illegal” (error production) branch of the tree, an error message can be produced, with an exact description of the error.

This example shows an error in a global identifier declaration.

```
int a = 6
```

In the source-code above the semicolon is missing, and the correct code is

```
int a = 6;
```

The error production below will catch this error and the following message will be printed to standard out.

```
Line 1: Missing semicolon in identifier declaration
int a = 6
```

10.1. Missing declaration and statement terminator

Every statement and declaration of an identifier must be ended by a semicolon, however it is a common mistake to forget the semicolon. By making alternative production rules (as shown below) the error can be caught and reported to the programmer.

The {error} (name of the alternative production) handles the case when a global identifier declaration is missing a semicolon, by not having a semicolon in its

production. This makes it possible, to catch the exact error when traversing the syntax tree and print an error message, when running into this production.

The {statement_err} has the same purpose and it has a similar production without the semicolon.

```
global_identifier_declaration →
          identifier_declaration semicolon
          | {error} identifier_declaration
          ;

statement_list →
          ...
          | {statement} statement semicolon
          | {statement_err}statement
          ;
```

This solution does not accept a single standing semicolon. Inserting it into the grammar, results in a shift/reduce conflict; this error is to be handled in error recovery. Further implemented error productions are listed below.

10.2. Invalid assignment operator and missing functions keyword

Use of an assignment operator from another programming language or simply a type error because there is another operator, which is written similarly, is also a common mistake. The missing **functions** keyword error production is also in one of these productions and is therefore also explained in this section.

Assignments exist as statements and as parts of variable and constant declarations.

Assignment statements, hold the risk of the programmer, uses a wrong token
In the {error1} production catches the case when an equal token (‘==’) is used instead of the assignment token (‘=’). A Pascal assignment (‘:=‘) instead of the correct assignment (‘=‘) is caught by {error2}.

```
assignment →
      identifier assign type_value
      | {error1} identifier equal type_value
      | {error2} identifier colonequal type_value
      ;
```

For the constant declarations, the {error1} handles the case, where the ‘:=’ token is used, and {error2} handles the situations, where the programmer forgets to assign a value to the constant in the declaration.

```
const_declaration →
      final data_type identifier assign type_value
      | {error1} final data_type identifier colonequal type_value
      | {error2} final data_type identifier
      ;
```

The variable declarations {as_error} is an assignment error, where the ‘:=’ token have been used instead of the correct ‘=’ token. The {func_error} checks if a procedure is declared before the **functions** keyword. Placing the production here, also have a side-effect, in that it also captures nested procedures.

```
var_declaration →
      {assigned}   data_type identifier assign type_value
      | {as_error} data_type identifier colonequal type_value
      | {unassigned} data_type identifier
```

```
        | {func_error} data_type identifier
                  l_par declarator_parameters? r_par block
        ;
```

10.3. Invalid procedure parameter delimiter

In both procedure declarations and procedure calls, parameters and parameter declarations are also something which varies among programming languages, both in syntactical structure and by the delimiting token, which separates the parameters. These productions are only made for the delimiter error. The two normal delimiters are comma and semicolon, and in this language comma is the correct one.

```
more_function_call_parameters →
      {correct} comma function_call_parameters
      | {error} semicolon function_call_parameters
      ;

more_declarator_parameters →
      {correct} comma declarator_parameters
      | {error} semicolon declarator_parameters
      ;
```

10.4. Invalid comparison operator

Another very common mistake in an expression is to use the assignment (‘=‘) operator instead of the comparison (‘==‘) operator, when making a comparison. The comparison operator also differ in many programming languages, which makes it a typical error for most programmers, this is also partly due to the everyday use of the ‘=’ token as comparison.

```
equality →
      {equal} equality equal relational
      | {error} equality assign relational
      | ...
      ;
```

The error productions are not checked or used by the compiler for anything else than to give error messages.

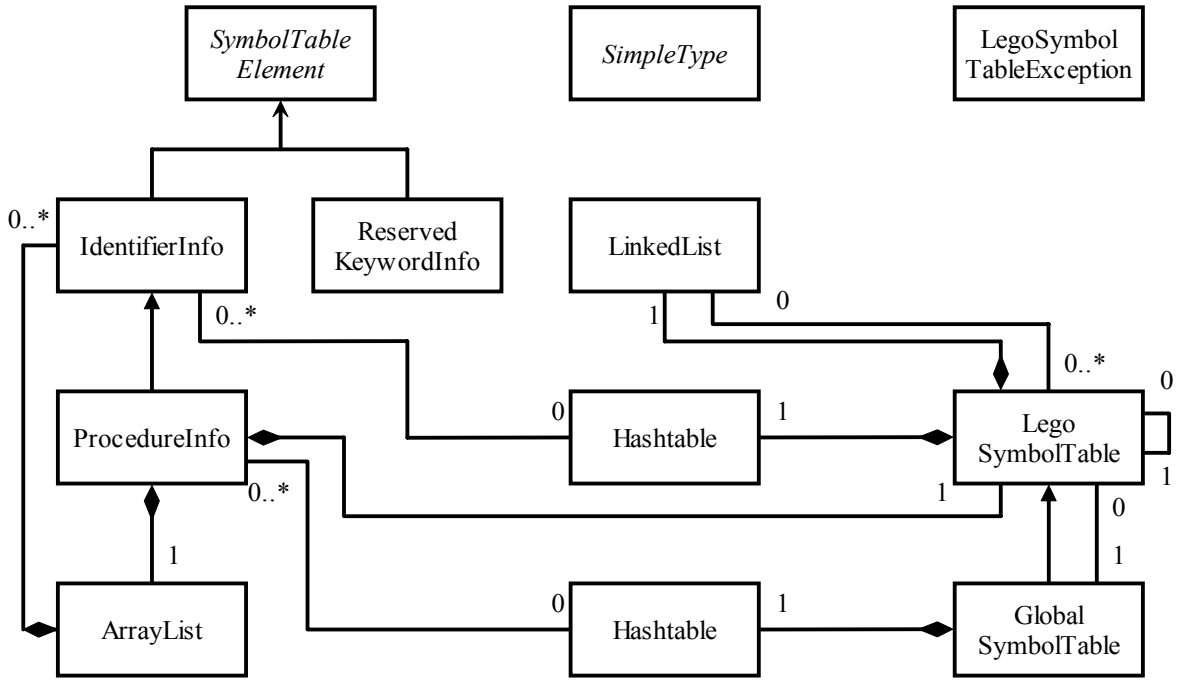
11. Symbol table

Since the SableCC framework does not provide a symbol table, a part of the project was to construct it. This section consists of three parts. First the description of the symbol table is discussed, where the table will be explained using an UML diagram followed by an example. The last section describes the practical use of the symbol table. The symbol table is designed to work in a language with lexical scope without nested procedures.

11.1. Description of the symbol table

The SableCC framework implies an object oriented perspective. The structure of the scopes will be kept in linked lists. Identifier information are stored and retrieved by hash tables.

This UML diagram illustrates the symbol table.



Figur 1 - UML diagram, Symbol table

The interface SymbolTableElement is a marker interface. It is implemented by the classes representing the contents of the symbol table, to have control over the type of the objects. Where a general type, representing the contents of the symbol table is needed, this marker interface is used, instead of the much more general object type.

The interface SimpleType contains only Java integer constants. These integers are used for setting and retrieving different information about identifiers. For examples see the list below.

```
public final int FINAL = 0;
public final int NONFINAL = 1;
public final int INTEGER = 2;
public final int BOOLEAN = 3;
public final int VOID = 4;
```

Example 4 – Content of SimpleType

The class ReservedKeywordInfo contains information about the reserved keywords. The information is only the lexeme representing the keyword. The idea of keeping information about the reserved keywords is that no identifier having a lexeme equal to a reserved keyword may be stored in the symbol table. But in this case it will not be necessary to initialize the symbol table with the reserved keywords, because the parser specification does not allow use of reserved keywords as identifiers.

IdentifierInfo class contains information about the identifiers representing the variables and constants. The lexeme, token, memory address, type (which simple type, final/non-final) and actual value are stored in instances of this class. The relation between the class IdentifierInfo and LegoSymbolTable means that no identifier knows which scope it belongs to.

ProcedureInfo is an extended class of IdentifierInfo. Instances of this class are used to hold information about identifiers representing procedures. Besides the inherited

information about lexeme, token and memory, information for return value, indication of procedure, an arguments list and an instance of LegoSymbolTable are stored. The arguments list is implemented as an ArrayList, which are a data type from the standard Java SDK 1.4 util package. In this ArrayList the arguments are stored sequential as they are declared. The instance of the LegoSymbolTable represents the actual procedure scope.

The LegoSymbolTable class represents scopes in procedure and all other scopes in between the tokens ‘{‘ and ‘}’ (called a block in the parser specification). For storage of the identifiers a Hashtable is used. Only identifier for variables and constants are allowed, because the language does not support nested procedures.

The Hashtable class is a data structure from the standard Java SDK 1.4 util package. The use of a hash table has the advantage that a request for an element stored in the hash table will be fast. In case of scopes inside a scope, these will be stored in the sequential order first declared scope first.

The data structure used for this storage is a LinkedList (from the util package). A scope always have a reference to the nearest outer scope. This is for lookup on identifiers. If not found in current scope the nearest outer scope is searched until global scope is reached. A reference to the global scope is also found in instances of LegoSymbolTable. For lookup on a procedure the lookup is always made in global scope, because this is the only scope they can be declared in (no support for nested procedures).

The class GlobalSymbolTable extended from LegoSymbolTable represents the global scope of a program. When the compiler is running only one instance of this class exists. The inheritance from LegoSymbolTable gives the ability to store global defined variables and constants. Like in LegoSymbolTable, where references to instances of IdentifierInfo are stored in a hash table, references to instances of ProcedureInfo are also stored in a hash table.

Exceptions of the exception class LegoSymbolTableException are thrown whenever an attempt is made to insert an illegal type (SymbolTableElement required) in a symbol table. This means that the insert methods in the symbol table have to be checked (try-catch), when they are used. This has no practical meaning when the compiler is up and running. During the construction of the compiler, no violation of the symbol table elements can occur.

To illustrate how the symbol table manages an input program with different levels of scopes, a diagram for the following sample code is shown below.

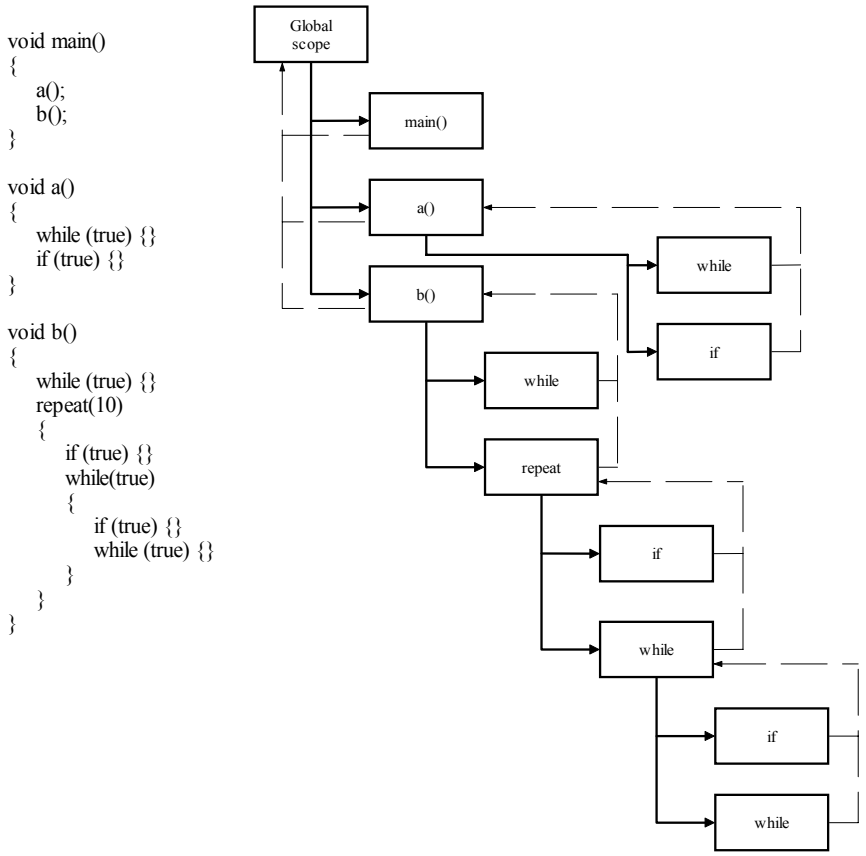


Figure 8 – Scope illustration

The dotted lines shows that all scopes knows their nearest outer scope. Not shown on the diagram, is that all scopes has a direct link to the global scope.

11.2. Practical use

Insertion, lookup and retrieval of identifiers in the symbol table distinguish between variables/constants and procedures. This gives the language the ability to have identifiers with the same lexeme, when one of them is a variable/constant and the other a procedure.

Creation of the symbol table is done just after the parser have constructed the AST. Because the symbol table is created before the semantic check and not in the same traversing it is possible to refer to identifiers before they are declared. The parser specification can not provide such a possibility. The grammar specification demands global variable and constants before procedures.

Two methods for look up on variables and constants are available. One is used for look up in the current scope and only the current scope. This feature is used by the semantic check when checking for re-declaration of variables and constants. The other method uses the reference to nearest outer scope, to check for non-local variables and constants.

When declaring variables and constants it is assumed that the left side of the assignment is correct. If the type of the right side does not match the left side, a default value of the correct type is assigned instead. This task is performed during the creation of the symbol table.

12. Semantic check

For semantic checks, the SableCC AST tree-walker and our own implementation of the symbol-table are at our disposal.

Operators and the stack

To check whether an operator has the correct operands, a stack is used. The following example will show how the stack is used and how the operands are checked for correctness.
The example shows a typical expression that evaluates to a Boolean.

```
((5 + 2) < 10) || (2 != 5)
```

Example 5 – Expression

The AST for this condition will look like this.

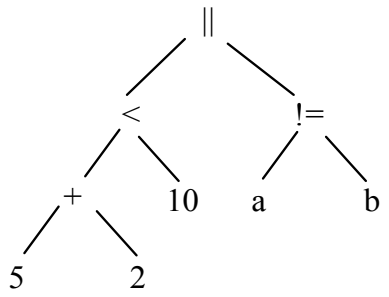


Figure 9 - AST for the example above

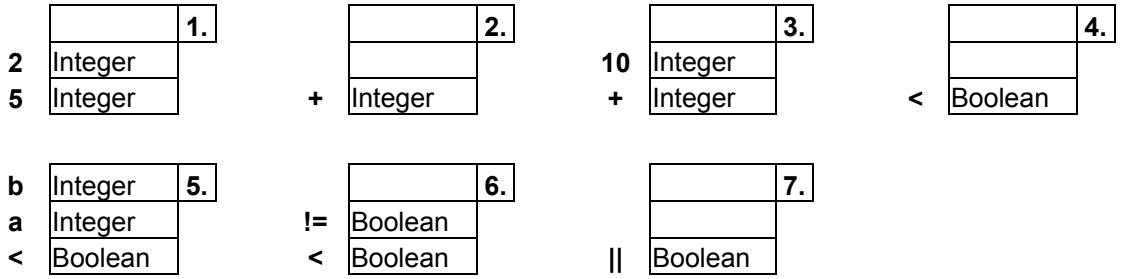


Figure 10 - The stack states while traversing the AST

1. When the tree-walker encounters the value “5” after going through the parent node, an Integer is pushed onto the stack. Then the tree-walker encounters the value “2” and yet another Integer is pushed onto the stack. The traversal now goes to the parent node.
2. When the tree-walker traverses from the “+” node to its parent node, two elements are popped from the stack, and are checked whether they can be used with this operator. Then an Integer is pushed onto the stack, because an expression with the “+” operator always evaluates to an Integer.
3. The tree-walker now traverses to the value “10” and pushes an Integer onto the stack.
4. When reaching the “<” operator, 2 elements are popped from the stack, it is checked whether both types are Integers and a Boolean is pushed onto the stack, as the “<” operator always evaluates to a Boolean.

5. The identifier “a” is now reached and its type is looked up in the symbol table and pushed onto the stack. In this case “a” and “b” is an Integer, so two Integers are pushed to the stack.
6. The “!=” operator pops two elements from the stack and compares them to see if they are of the same type, in this case they are and because of the “!=” operator always evaluates to a Boolean, and a Boolean is pushed onto the stack.
7. The last node in the example will evaluate the last two elements of the stack into a new Boolean and the semantic checking of the complete condition is now finished.

Errors are thrown when checking the types. If they are not valid, an exact error message can be thrown, with information about the operands that causes the error.

Selections and loops
There are two types of loops and one type of selection; the “While” loop and the “Repeat” loop and the “If” selection.
The “While” loops and “If” selections are semantically alike, they both need a valid condition to work, however the “Repeat” loop needs an expression. A condition should always evaluate to a Boolean whereas an expression should always evaluate to an Integer.

Using the implementation of conditions from above, (the implementation for expressions is exactly the same) it is possible to check whether a “While” or an “If” has a condition or if it is a “Repeat”, an expression.

Return values
It can be seen, when traversing into any function, whether the function has a return statement or not, because the grammar differs the function block in two different parts. This makes it possible to throw an error if a function with the return type “void” has a return statement and at the same time it is possible to throw an error if the function has a return type and a return statement is not present in the function scope.

Because return values are evaluated like conditions and expressions, the semantic check only needs to be implemented in these to ensure correct evaluation.

Constants and variables
The stack is used to check whether constant and variables have the correct types. Whenever a constant or variable is declared, an element is pushed onto the stack for the type and a type for the condition, expression or function call.

To check whether a constant is re-declared or if its value is changed, the symbol table is needed. Whenever a constant declaration is encountered, the constant is looked up in the symbol table and checked whether it is already declared in the current scope. If a variable declaration is encountered, it is also checked in the current scope to see whether it is already present. If a variable is being assigned, it is checked whether it is present in the symbol table, and if it is present its new value is written to the symbol table. If the right side of the assignment is another variable or a constant, it is also checked whether or not this side is present in the symbol table and assigned.

Procedure calls
When calling a procedure, the first check that is done is to check whether the procedure exists, if the procedure exists, all parameters, if there is any, are checked both for correctness of the types and that the number of parameters in the procedure call matches the number of parameters in the procedure.

The symbol table holds information about the number of parameters for the procedure, with this information, when traversing out of each parameter in the procedure call, it can be established whether the types are correct. When traversing out of the procedure call itself, it is checked whether or not there are more parameters.

The reason for using a stack instead of specific action code at each node, is that using the latter, results in a weakening of the framework, because any changes in the grammar, would require a complete rewrite of perhaps all the productions action code. Using the stack instead, strengthens the integrity of the framework, as any modifications to the grammar, will only require modification of the surrounding productions.

13. Changes and new goal

Since it turned out that the RCX-unit was very limited in its computational capabilities, the starting goal of this project has changed dramatically. Our initial goal was to write a compiler, capable of translating our defined language to RCX-unit understandable hex code.

The problem was that the RCX-unit did not have a dynamic allocating memory in which a stack could exist. Instead the RCX-unit had reserved memory space for exactly 32 global variables, 10 tasks and 8 subroutines. Because of this, return values and parameters are not supported in routines. Local variable does not exist, and recursive procedures are not possible. This made it impossible to show the abilities of how to control a stack during code generation, which is an important part of the theory.

Further more it was not possible to use some of the 32 memory blocks as a stack. This is because all variables are assigned a block at compile time. Figure 11 shows what at first glance might look like a solution with a very limited stack. First block holds a stack pointer which is followed by the programs global variables. Rest of the blocks is used for the stack. However, as it is not possible to use a block as a pointer to another block, that was not an option.

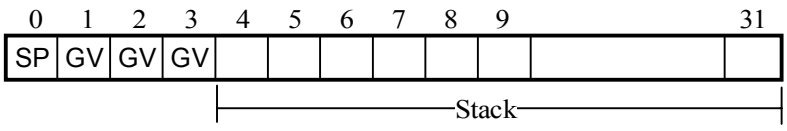


Figure 11 – Example of a stack implementation for the RCX-unit

This problem forced us to make a though decision. Should we continue on the same track and make a target language that was actually useful and understandable by the RCX-unit, and skip the important stack problem. Or should we change the target to something not understandable by the RCX-unit, but would give us the “missing” capability, and problem, which had to be handled.

Since the main goal of this project, is to show understanding and knowledge of compiler construction, we decided in collaboration with our supervisor, to change the target code to include the stack.

Instead of hex code as our target language, we decided to change it to assembler code. Changing the target to an assembler language, which are translatable to an executable file, and forgetting all about the RCX-unit, was not in our interest. This would render parts of the project useless, as a lot of documentation for the RCX-unit was already made. Therefore it was desirable to further extend our language, which would not be translatable into a functioning program.

The latter case, had the better appeal, and to keep some of our initial purpose of the project we decided to invent and design a more powerful next generation language for the RCX-unit.

NQC is another language already developed for the RCX-unit. It is generating hex code which can be uploaded to the RCX-unit. During compilation NQC is generating intermediate code. This code is very similar to assembler code.

NQC source code	Intermediate assembler code	RCX compatible hex code

int aaa;
int bbb;
 	
task main()
{
  start calculations;
  if (aaa == 50)
  start RcxControl;
}
 	
task RcxControl()
{
  OnFwd(OUT_A);
  OnFwd(OUT_B);
  Wait(400);
  Off(OUT_A+OUT_B);
}
 	
task calculations()
{
  aaa = 10;
  bbb = aaa * 5;
}
	*** Var 0 = aaa
	*** Var 1 = bbb
	
	*** Task 0 = main
	000 pwr ABC, 7
	004 dir ABC, Fwd
	006 start 2
	008 chkl 50 != var[0], 18
	016 start 1
	
	*** Task 1 = RcxControl
	000 dir A, Fwd
	002 out A, On
	004 dir B, Fwd
	006 out B, On
	008 wait 400
	012 out AB, Off
	
	*** Task 2 = calculations
	000 setv var[0], 10
	005 setv var[1], var[0]
	010 mulv var[1], 5
		
		13 07 02 07
		e1 87
		71 02
		95 82 00 32 00 00 04 00
		71 01
		
		e1 81
		21 81
		e1 82
		21 82
		43 02 90 01
		21 43
		
		14 00 02 0a 00
		14 01 00 00 00
		54 01 02 05 00

Figure 12- NQC's intermediate assembler code compared to NQC source and RCX byte code.

Instead of generating hex code, our target language will be heavily inspired by NQC's intermediate assembler code. We will however make some changes to gain the ability of a memory in which we can have a stack. The 32 spaces where global variables previously could exist are now used as registers. We will further extend the assembler language with commands to address the new dynamic memory area. In addition to the dynamic memory area, we will create a static data area, where global data known at compile time can be placed.

With these changes we will create a more powerful target environment and we have kept our initial goal of writing to the RCX-unit. The consequence of our modification to the RCX-unit is that it is harder to test if the result of the generated target code is

correct. First of all we can not compare the target code with a similar target code generated by NQC. Secondly the changes to the target language mean that it is not possible to upload the target code to the RCX-unit any longer. Thereby we can not test the generated code with the RCX-unit. What we instead can do is to show our understanding of code generation, assembler and our ability to solve the issues involved.

14. Storage allocation

Before code generation we need to consider some storage allocation techniques. One of the main goals in this project is to design a language which corresponds to the Java programming language as closely as possible. In Java the opportunity lies to use recursive procedures and that is why we have chosen to implement this feature into the language.

As mentioned earlier (in the former chapter) this decision causes some design problems with the RCX-unit. The memory blocks in the RCX-unit are unable to refer to each other, so even an implementation, with a minimum amount of memory (requiring a stack pointer) was not possible. An all round description of recursive procedures and how exactly we will handle this problem will be described in the following.

To illustrate how recursive procedures work we have used some (simplified) examples from the textbook [ASU88].

A subdivision of the run-time memory is used for keeping track of procedures. This structure is known as a control stack. This structure will be described in details and how the stack structure is designed and how it is used in practice.

Some of the memory needed at run-time is already known at compile-time. This portion of the memory can be allocated statically. A detailed description on this subject will follow.

14.1. Recursive procedures

An execution of a procedure is also referred to as an activation of a procedure. Recursive procedures are defined as being several activations that may be alive at the same time. More specific it can be said that a procedure is recursive if a new activation occurs before an earlier activation of the same procedure has ended. This is described in the following example.

```
...
private void QuickSort(int low, int high)
{
  int i;
  if (low < high)
  {
    i = Partition(low, high);
    QuickSort(low, i - 1);
    QuickSort(i + 1, high);
  }
}
```

Example 6 - Part of a quicksort algorithm implemented in Java

In the example above the QuickSort procedure is first entered using two numbers, for example the highest and the lowest position of an array. Then the Partition procedure is entered and immediately leaved again. QuickSort is then entered again before the first activation has ended; this means that this procedure is recursive.
It is important to note that a procedure needs not to call itself directly to be recursive. For example a procedure “p” can call another procedure “q”, which then calls the procedure “p” recursive.

Another way to describe this is by using an activation tree.

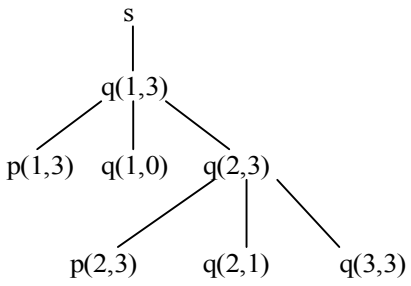


Figure 13 - An activation tree laying out the quicksort example

The activation tree, where the root represents the main program, shows each activation of a procedure as a node. A node is only a parent to another node if and only if the control flows from the parent node to the child node. The leftmost children of a parent with multiple children will be visited first.

A depth-first traversal of the activation tree, starting at the root and visits every node in a left-to-right order. This corresponds to the flow of control in a program, also referred to as a control stack.

A control stack is used to keep track of all live activations. Whenever a node is reach and the activation begins, an activation record[15] is pushed onto the stack. When the activation ends the activation record is then being popped of the stack.

Storage organization
It will be explained in this section how run-time storage is managed by the compiled program. The block of storage should hold the generated target code, data objects and a control stack to keep track of the procedure activations.

The size of the generated code and the data objects (we here refer to our global variables and constants) are both known at compile time. Both these can be placed in the statically determined area of the memory. It is a good idea to allocate as many data objects statically as possible, because these can be compiled into the target code. As mentioned before a stack is needed to control the procedure activations so this has to be allocated dynamically. Furthermore a separate area of the run-time memory, called a heap, holds all other information that the compiled program may use dynamically.

[15] An activation record is a block of information in memory needed by a single activation. An activation record consists, among other, of returned values, actual parameters and local data.

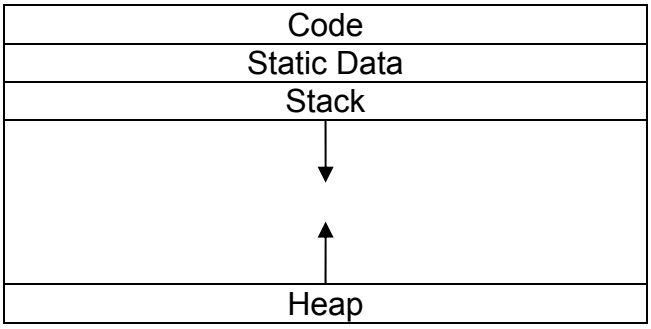


Figure 14 - Typical subdivision of run-time memory

The stack and heap can change as the program executes, this is illustrated by the arrows. The heap and the stack lie in each end of the memory so they can grow toward each other as needed. In this way the stack and the heap do not need to worry about overlapping, unless no more memory is available anyway. The stack in this model is also known as a downward-growing stack, because whenever an activation record is pushed on to the stack, it grows downwards.

A heap de-allocation does not necessary occur in a last-in first-out order. De-allocation may happen in any order as necessary. This means that over time alternate areas of the heap are freed and some may be in use. A heap manager handles the free space in between used space. This may cause some time and space overhead, but is necessary for an effective use of the memory space.

Modification to the RCX-unit
The memory restrains of the RCX-unit causes some major modification to the RCX-unit.

A description on how we extend the RCX-unit (fictitiously) with more memory which then can be allocated dynamically is illustrated below. It is in this memory the stack resides.

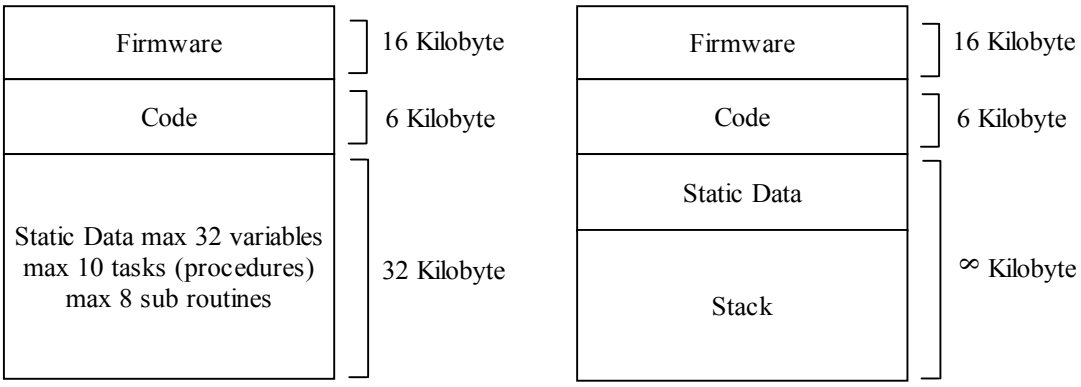


Figure 15 – To the left is the RCX-unit as we know it and to the right is our extended RCX-unit

The 32 available memory blocks is now used as registers like the ones on a CPU. It was decided to extend the RCX-unit with an infinite[16] amount of memory, which can then hold code, static data and a stack for managing activation records.

[16] Naturally there is no infinite amount of memory, but we avoid setting a limit.

Furthermore it is decided not to implement a heap to our fictitious memory. This means that it must be certain that the values of local names are retained when activation ends and a called activation must not outlive its caller.

A heap is usually used for data structures like dynamic arrays and objects so there is no reason to implement this into the modification of the RCX-unit.

The expansion of the memory for the RCX-unit results in a broader target code, which have to manage the push and pop procedures for the stack. This extension is described in the chapter describing the code generation.

14.2. Static allocation

The static storage area contains the code for the program and the activation records for all the procedures in the program. The static storage area is organized, so that the code is put in one memory block, and the activation records are placed in another one afterwards, as illustrated in the figure below.

```
void main() {
            int time = 5;
            run( 5);
}
void run( int value) {
            goFwd( 5);
}
```

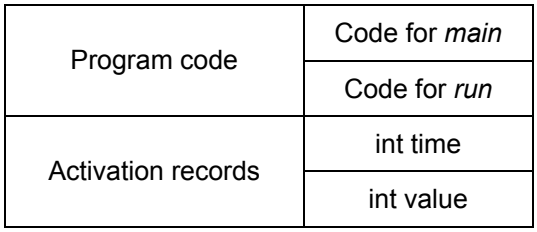


Figure 16 - Activation records and program code allocated statically

When making the translation into target code, the amount of memory required for immediate execution can be determined by looking at the variables data type. This information is then used to create a layout of the storage for all the data objects at compile time.

In relation to our language, static allocation have some limitations to it, such as all data objects must be known at compile time and recursive procedures are not possible, since all activations of a procedure uses the same activation record.

Data objects at compile time

As the language does not allow self defined types, the size of all the data objects is known. Since the data objects are of a fixed size, the activation records are made of a fixed size as well, so this is also known.

This makes is possible to set an offset from the bottom/top of the activation record to each data object inside. So at compile time, when the activation records have been created and all offsets determined, we can fill in the addresses for the objects in the activation records.

Static allocation and recursive procedures

When it comes to recursive procedures however, we will not be able to support them, without additional run time memory allocation. Therefore stack allocation is employed to open for the possibility of recursive procedures. Stack allocation will be introduced in the following section, along with a description of how it will operate during run time.

The static allocation will still hold the program code, but because we need to make recursive calls to procedures, we will not be able to determine the number of

activation records needed. This problem requires a well defined split between the static memory and the stack memory for the procedures.

We have two choices in this situation:

1. The activation record for the first instance of any procedure is stored in static memory and for any recursive call, the activation records are pushed onto the stack.
2. All activation records are pushed onto the stack and only global variables are kept in static memory.

The first choice would have the advantage of allocating as much memory minimum required, for run time. But also requires the compiler to keep track of every procedure call to see it is the first call or a recursive call, to determine if a new activation record should be created.

The second choice on the other hand benefits from always having a new activation record created for any procedure call, and thereby making a simpler compiler. This on the other hand, will result in more operations on the stack.

Our decision landed on the second choice as we found it better to have a consistent way of using the activation records and generating the code.

14.3. Stack allocation

As mentioned earlier the activation records are pushed and popped on and off the stack, whenever a new activation starts and ends. Local variables for each procedure are stored in the activation record. These variables are deleted when the activation ends.

The size of all activation records in our language are known at compile time. In other words there is no dynamic length data like arrays in our language. The conclusion to this is that there is no need for at pointer structure, that points to somewhere in the heap.

The way activation records are pushed and popped on and of the stack is outlined in the following diagram. The transaction from the activation tree in Figure 17 to a stack of activation records are illustrated for at small portion of the activation tree.

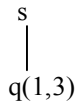
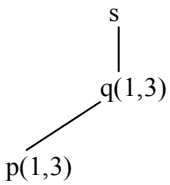
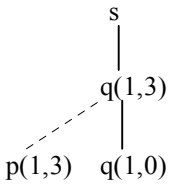
Activation tree	Stack	Remarks
	**s** ... **q(1,3)** low 1 high 3 i ?	The first activation record to be pushed onto the stack is that of procedure s. This procedure can be considered to be the main method of this program. Secondly the next activation record to be pushed onto the stack is the q(1,3).
	**s** ... **q(1,3)** low 1 high 3 i ? **p(1,3)**	q(1,3) calls the p(1,3) procedure and the activation record for p(1,3) is pushed onto the stack.
	**S** ... **q(1,3)** low 1 High 3 i ? **q(1,0)** low 1 high 0 i ?	After p(1,3) has terminated, the activation record for p(1,3) is popped from the stack. The control then returns to q(1,3) which then calls the q procedure recursively and the activation record for procedure q(1,0) is pushed onto the stack.

Figure 17 - Downward growing stack

A stack pointer is a register, which always marks where the top of the stack resides. Whenever an activation record is pushed onto the stack, the stack pointer is incremented with the size of that record. Respectively the stack pointer is decremented again when the activation record is popped from the stack.

A calling sequence is used to allocate an activation record and enter information into it. The calling sequence is divided between a caller and a callee. The procedure which is used to call another procedure is the caller and the called procedure is the callee. In the above example this means that q(1,3) is the caller to p(1,3) and q(1,0).

To make parameters from the calling procedure available for the called procedure, these are passed down into the activation record of the called procedure.

15. Code generation

When the syntactic and semantic checking has been completed successfully, the code has been proven to be free of errors, which would make the compiler unable to

generate target code. Logical errors can still exist, as no checking is performed to capture these.

We will be skipping any intermediate code generation, since our focus is on neither optimizing nor multiple targets, but simply the whole process compiling the source code to target code.

The new memory design is introduced in this chapter. This design will enable the RCX-unit to handle recursive calls. To make the assembler code easier to understand, this chapter will contain a detailed description of how function calls and recursive calls are handled. A detailed description of the construction of loops and conditionals in assembler code is also to find. In the compiler generated code, comments are added by the compiler using "::" to separate the generated code and comments. This makes it easier to read and understand the code.

15.1. Redesigning the memory of the RCX-unit

The original 32 memory blocks for holding global variables have been converted to regular registers similar to the ones in a CPU (named $r1, $r2, …, $r27). In addition the following specialized registers have been introduced:

- A return-value register (named $rv). This register is to hold return values when leaving procedure scopes.
- A register for holding the engine selector value (named $rr), specific for RCX-commands.
- A register for holding the stack pointer offset (named $ro).
- A register for holding a loop counter (named $rc).
- A register for holding the value for the stack pointer (named $sp).

Storage space for procedures are named p0, p1, …,p*n*. Storage space for static variables has been implemented (named g0, g1, …,g*n*) as previously described in chapter 14.2. There is at this point no maximum count for procedures and static variables, however this could easily be implemented. Below is the new memory design illustrated.

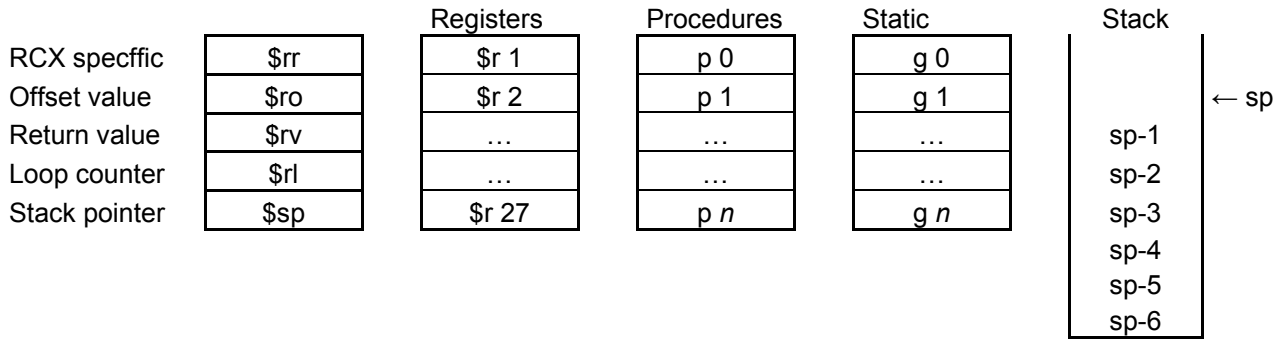


Figure 18 - The new memory design

Further more a stack has been implemented. This stack is to hold the activation records for procedures and also function as a temporary storage for handling conditions, expressions and terms. It is possible to access elements put on the stack in the same scope or previous scopes. The top of the stack is accessed via the stack pointer (sp), and values in the stack are accessed via an offset from the stack pointer.

To handle the new memory design, the instructions below have been implemented. Later on in this chapter the use of the memory design and instructions will be further described and put into use.

Name	Variabels 1	2	3	Description
Dec	<register>			Decrements a register by 1
Inc	<register>			Increments a register by 1
Copy	<fromAddress>	<toAddress>		Copies the content from one memory space to another.
Setv	<address>	<value>		Saves a value in a specific memory space.
Add	<toAddress>	<register1>	<register2>	Adds the <register1> with <register2> and puts the result in <toAddress>
Lt	<toAddress>	<register1>	<register2>	Less than: Compares <register1> with <register2> and puts the result in <toAddress>. 1 if true and 0 if false.
nEqual	<toAddress>	<register1>	<register2>	Not equal: Compares <register1> with <register2> and puts the result in <toAddress>. 1 if true and 0 if false.
Or	<toAddress>	<register1>	<register2>	Or: Compares <register1> with <register2> and puts the result in <toAddress>. 1 if true and 0 if false.

Figure 19 – Assembler instructions

15.2. Processing conditions, expressions and terms

Conditions, expressions and terms are handled in much the same way as in the semantic check, tree-walkers are used to traverse the AST and the action code is defined for the productions via the in- and out methods for the production. Below is shown how a condition is processed and how values are copied between the stack and the registers. The condition evaluated and its AST:

```
( ( 5 + 2 ) < 10 ) || ( 2 != i )
```

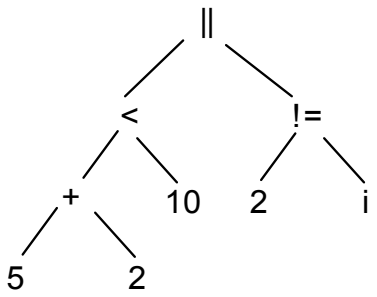


Figure 20 - AST for the above example

The AST is traversed until the value 5 is reached. Code for the RCX-unit is generated so that 5 will be pushed onto the stack in the RCX-unit and the tree is traversed until an out-method on an operator or a value is met. In this case code is generated to push the value 2 on to the stack.

```
Setv      sp        5
Inc       $sp
Inc       $ro
Setv      sp        2
Inc       $sp
```

```
Inc         $ro
```

The created code above pushes the value 5 onto the stack, both the registers for the stack pointer and offset value are incremented. Then the value 2 is pushed onto the stack, and again both the registers for the stack pointer and offset value are incremented.

The tree-walker then reaches an out-method on an operator, in this case “+”. Code is generated to copy the pushed values from the stack to the registers. Here they are added and the return value for the expression is pushed on to the stack.

```
Copy        sp-1        $r1
Copy        sp-2        $r2
Sub         $sp         $sp         2
Sub         $ro         $ro         2
Add         $r3         $r2         $r1
Copy        $r3         sp
Inc         $sp
Inc         $ro
```

Now the stack only holds the return value (7) from “+”. As the AST is traversed the value 10 is pushed onto the stack and the relational “<” is then evaluated by moving the two values from the stack into registers and performing the evaluation.

```
Setv        sp          10
Inc         $sp
Inc         $ro
Copy        sp-1        $r1
Copy        sp-2        $r2
Sub         $sp         $sp         2
Sub         $ro         $ro         2
lt          $r3         $r2         $r1
Copy        $r3         sp
Inc         $sp
Inc         $ro
```

The stack only holds the return value (true) from “<”. As the AST is traversed the value 2 is pushed onto the stack. When the tree-walker reaches an identifier, the value of the identifier is fetched form its original memory place and pushed onto the stack. The equality “!=” is reached and the evaluation is performed.

```
Setv        sp          2
Inc         $sp
Inc         $ro
Copy        ???         sp      ::??? address for i
Inc         $sp
Inc         $ro
Copy        sp-1        $r1
Copy        sp-2        $r2
Sub         $sp         $sp         2
Sub         $ro         $ro         2
nEqual      $r3         $r2         $r1
Copy        $r3         sp
Inc         $sp
Inc         $ro
```

The stack now holds two values as the tree-walker reaches the out procedures of the condition “||”. Again the two values are moved from the stack into registers and the evaluation is performed.

```
Copy        sp-1        $r1
Copy        sp-2        $r2
Sub         $sp         $sp         2
Sub         $ro         $ro         2
Or          $r3         $r1         $r2
Copy        $r3         sp
Inc         $sp
Inc         $ro
```

Finally the stack holds the result (true) of the condition and this can be popped and used.

15.3. Labels

The RCX-unit does not support labelled jumps. Instead jumps are made to offsets relative to the start of the code. As a consequence of this the intermediate NQC generated assembler uses line numbers as jump destinations. This is however not general assembler syntax, moreover it is a lot harder to generate and control the correctness of such code. As a consequence of this, this language will support labels and jumps to labels. See below for examples.

15.4. Repeat loop

The repeat loop is a loop running the number of time specified. In the RCX-unit, one of the registers has been reserved as a loop counter, this is named $rc. Just before a repeat loop is entered, the amount of times the loop will run is put in this register. Each iteration will decrease this value by one. When the counter reaches zero the loop will stop its iteration. Below is a code example using a repeat loop:

```
. . .
repeat  ( 3 )  {
           . . .
}
. . .
```

First the code to put the value 3 in register $rc is generated. Then a label is generated to mark the entrance of the loop. At that point code generation for the body is performed. Finally if register $rc is decreased and a jump is done back to the loop entrance. When the loop counter has reached zero a jump is made paste the code body.

```
           . . .
           Setv       $rc        3
Begin1:    Jez        Out1
           . . .
           Dec        $rc
           Jump       Begin1
Out1:      . . .
```

Since a repeat loop can be contained in another repeat loop, the value of $rc is backed up as the first instruction when generating code for nested repeat loops. This is done by pushing the old value of $rc on the stack. Upon exiting an inner loop, the old value of the $rc is restored. Below is a code example with a nested repeat loop.

```
. . .
repeat  ( 3 )  {
           repeat ( 5 )  {
                      . . .
           }
}
. . .
```

This will be generated the following.

```
           . . .
           Setv       $rc        3
Begin1:    Jez        Out1
           Copy       $rc        sp
           Dec        $sp
           Dec        $ro
           Setv       $rc        5
Begin2:    Jez        Out2
```

```
            . . .
            Dec         $rc
            Jump        Begin2
Out2:       Copy        $sp-1       $rc
            Inc         $sp
            Inc         $ro
            Dec         $rc
            Jump        Begin1
Out1        . . .
```

15.5. Conditional statements

We have chosen to implement two conditional statements; while and if. While executes a block of statements as long as the condition evaluated is true. If-statement only executes a block of statements once if a condition is true. The if-statement can be used together with else. In this case if the if-statement is false, the else block of statements will be carried out and vice versa. In the chapter above a condition is processed and a Boolean (value 0 or 1) is left on the stack. This result is used to determine whether or not the if-statement block is to be carried out. Below are examples of using the while-, if- and if else-statements. To handle these operations, the following assembler instructions are used.

	Variabels			
Name	**1**	**2**	**3**	**Description**
Jez	<address>	<label>		Jump to lable if the value at memory address is equal 0.
Jnez	<address>	<label>		Jump to lable if the value at memory address is not equal 0.
Jump	<lable>			Unconditional jump to label.

Figure 21 - Assembler instructions

Code example using while:

```
. . .
int i = 5;
. . .
while ( i < 10){
            i = i + 1; //While-block
}
. . .
```

First the code for assigning i to the value 5 is created and the symbol table is updated with the address of i. Secondly the value of i and the value 10 are pushed on to the stack, copied to registers, processed and pushed onto the stack. To make the code more understandable, stack pointer and offset operations have been left out.

```
            Setv        sp          5
while1:     Copy        $sp-1       $r1
            Setv        sp          10
            Copy        $sp-2       $r2
            Lt          $r3         $r2         $r1
            Copy        $r3         sp
```

At this point a “0” (= false) is on top of the stack. The condition is evaluated via a register and since the condition is false a jump to label out1 is not made. When the condition is found to be true, a jump to out1 is made.

```
            Jnez        $r1         out1
            Copy        sp-2        $r1         ::While1-block
            Add         $r1         $r1         1
            Copy        $r1         $sp
            Jump        while1
out1:       . . .
```

The if-statement is much the same except for the return jump to while1. The if-statement combined with else results in a conditional jump and a forward jump.

Code example using if with else:

```
. . .
if ( <condition> ){
            . . .       // if-block
}
else {
            . . .       // else-block
}
. . .                   // next statement
```

Code generated by the compiler to handle if with else:

```
            . . .
            Jnez        $r1         else1
            . . .                               :: if-block
            Jump        out1
else1:      . . .                               :: else-block
out1:       . . .                               :: next statement
```

If the evaluated condition is true, the if-block is executed and after this a jump is made to the next statement. If the condition is false a jump is made to the else-block, this is executed and afterwards the next statement is reached.

15.6. The function declaration problem

A problem occurred in code generation, allowing the main function to be declared before other functions, makes it possible to use these functions before they have been assigned a memory storage, see the example below.

```
void main(){
     . . .
     a();
}

void a(){
     . . .
}
```

This problem could have been solved by demanding that functions had to be declared before use. The solution was to assign functions their memory location while updating the symbol table. Alternatively the AST had to be traversed to perform storage allocation before code generation could begin.

15.7. Handling function activation records and return values

An important thing to handle is activation records. To do this properly the stack acts as the broker between the caller and the callee, see below for example code.

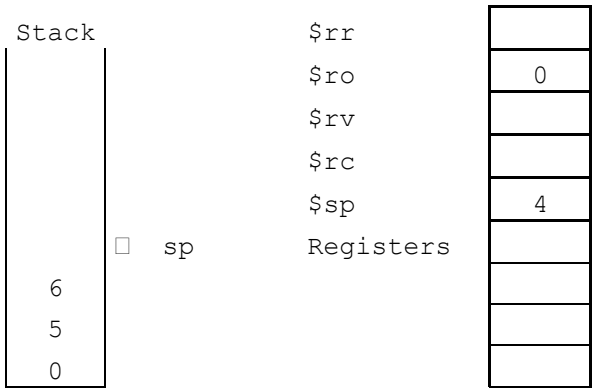
```
void main(){
     int x = a( 5, 6);
}

int a( int b, int c){
     return b + c;
}
```

When a procedure is called, the offset value for the calling procedure is pushed onto the stack, and the offset value is reset. The called procedure parameters are pushed onto the stack and when entering the callee function associates its declaration

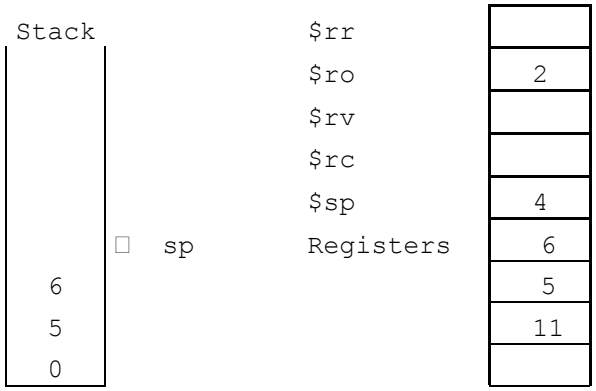
parameters in the procedures symbol table, with the address of the values on the stack pushed by the calling procedure. Below is the code generated and content of the stack and the registers shown.

```
Copy    $ro    sp
Inc     $sp
Setv    sp     5
Inc     $sp
Setv    sp     6
Inc     $sp
Setv    $ro    0
```



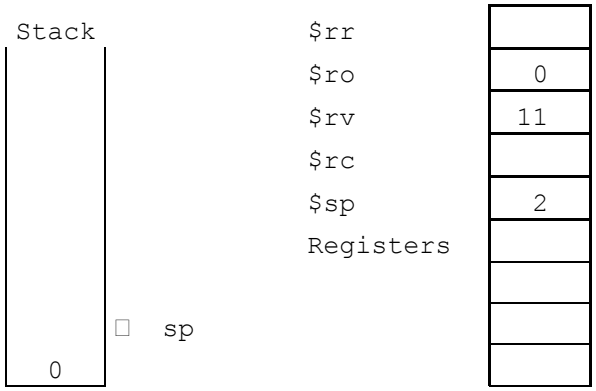
As shown earlier the expression in processed by copying the values of the two identifiers on the stack and pushing the result onto the stack.

```
Add     $ro    $ro    2
Copy    sp-1   $r1
Copy    sp-2   $r2
Add     $r3    $r1    $r2
```



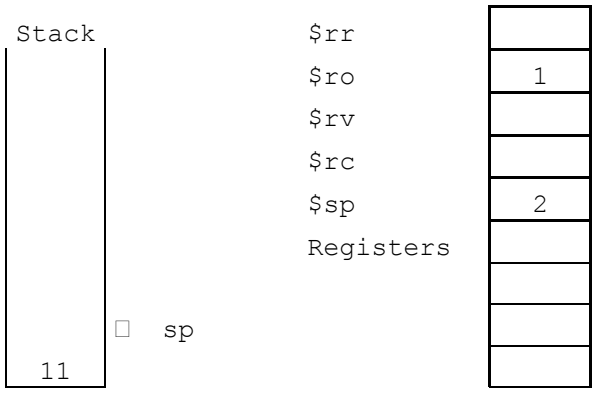
Finally the return value is popped from the stack and into the return value register and the procedure terminates by deleting its activation records.

```
Copy    $r3    sp
Inc     $sp
Inc     $ro
Copy    sp-1   $rv
Dec     $sp
Dec     $ro
Sub     $sp    $sp    2
Sub     $ro    $ro    2
```



When returning to the calling procedure the return value is pushed onto the stack for further use, in this case an identifier is declared and it is assigned to the address of the return value on the stack, since the value is already on the stack, the symbol table is updated with the address of the value.

```
Copy    sp-1   $ro
Dec     $sp
Copy    $rv    sp
Inc     $sp
Inc     $ro
```



15.8. Recursive calls

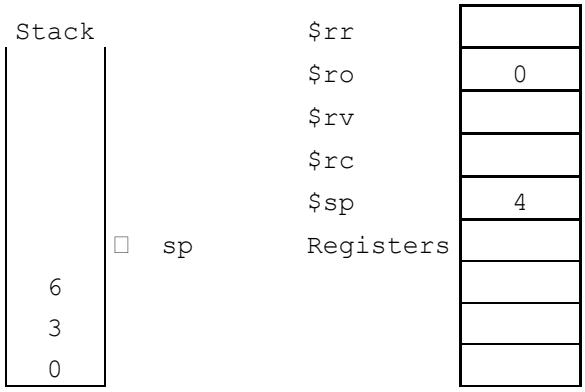
As shown in the above example the stack is used to store activation records and variables while in the procedure scope. In the example below the procedure a() calls itself recursively until a condition is met, then the procedure returns a value back through the calling procedure and finally x is assigned to the result of the recursive call. We will in this chapter show how operations done on the stack and in the registers. Some of the assembler code shown in this chapter is not generated for the procedures to work, but shown to understand the operations done on the stack and in the registers.

```
void main(){
     int x = a( 3, 6);
}

int a( int b, int c){
     b = b - 1;
     int d = 10;
     if (b != 0){
          d = d + a( b ,c);
     }
     return d;
}
```

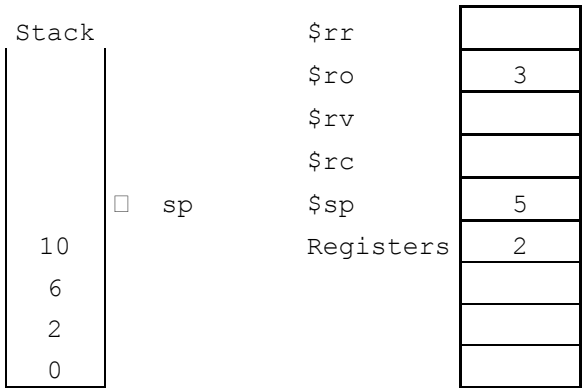
When a procedure is called, its parameters are pushed onto the stack and when entering the called procedure, the declaration parameters in the procedures symbol table are associated with the address of the values on the stack.

```
Copy    $ro    sp
Inc     $sp
Setv    sp     3
Inc     $sp
Setv    sp     6
Inc     $sp
Setv    $ro    0
```



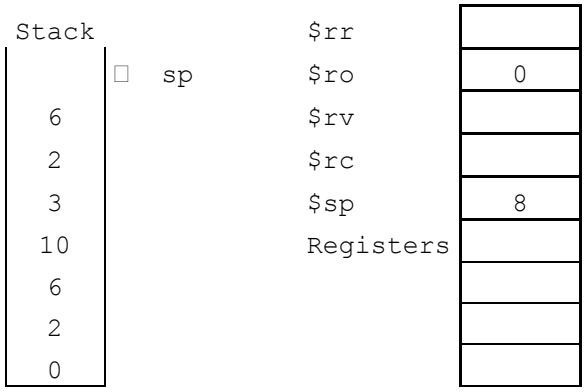
The value of b is copied to the registers, decremented and the new value is copied back to b's position in the stack.

```
Add     $ro    $ro    2
Copy    sp-2   $r1
Dec     $r1    1
Copy    $r1    sp-2
Setv    sp     10
Inc     $sp
Inc     $ro
```



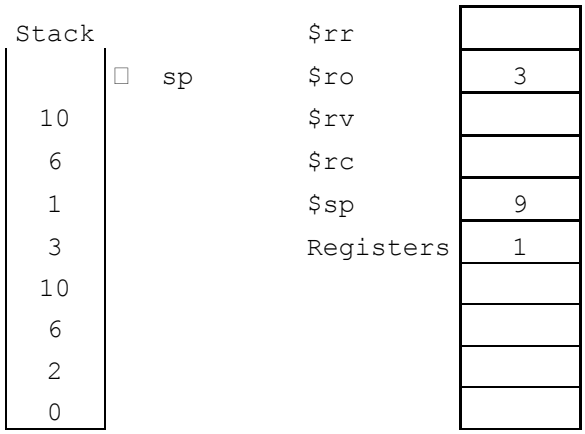
The condition `if (b != 0)` is handled here, for at detailed description of how this is done, see chapter 15.5. If the condition is met, the function a() is called again. First the offset is pushed onto the stack and the offset is reset, then a's activation parameters are pushed.

```
Copy    $ro    sp
Inc     $sp
Setv    $sp    2
Inc     $sp
Setv    $sp    6
Inc     $sp
Setv    $ro    0
```



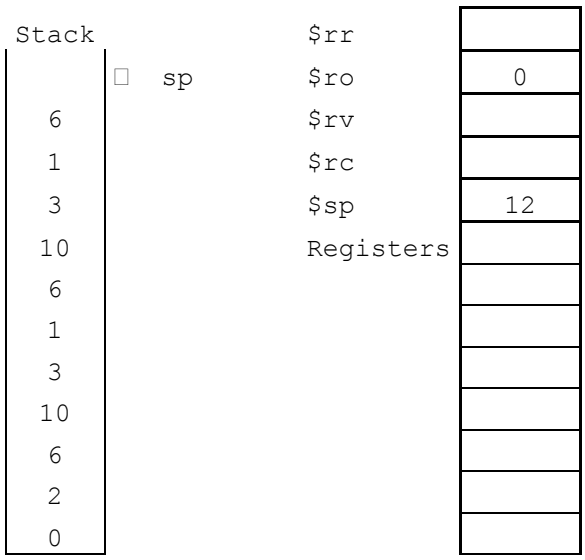
The value of b is copied to the registers, decremented and the new value is copied back to b's position in stack.

```
Add    $ro    $ro    2
Copy   sp-2   $r1
Dec    $r1    1
Copy   $r1    sp-2
Setv   sp     10
Inc    $sp
Inc    $ro
```



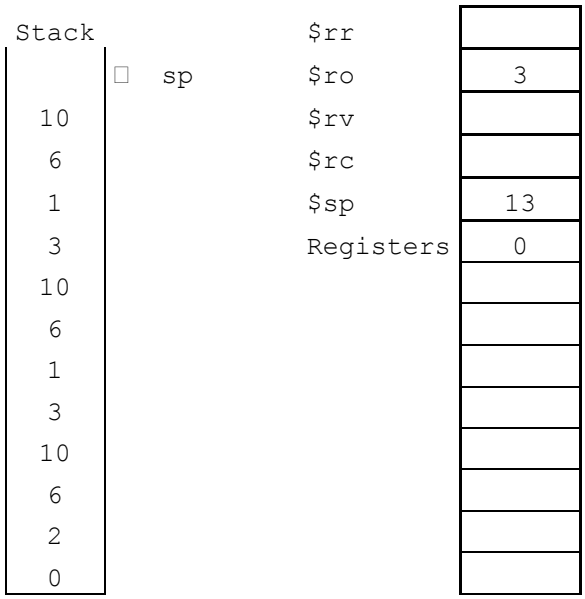
If the condition is met, the procedure a() is called again. First the offset is pushed onto the stack and the offset is reset, then a's activation parameters are pushed.

```
Copy   $ro    sp
Inc    $sp
Setv   sp     1
Inc    $sp
Setv   sp     6
Inc    $sp
Setv   $ro    0
```



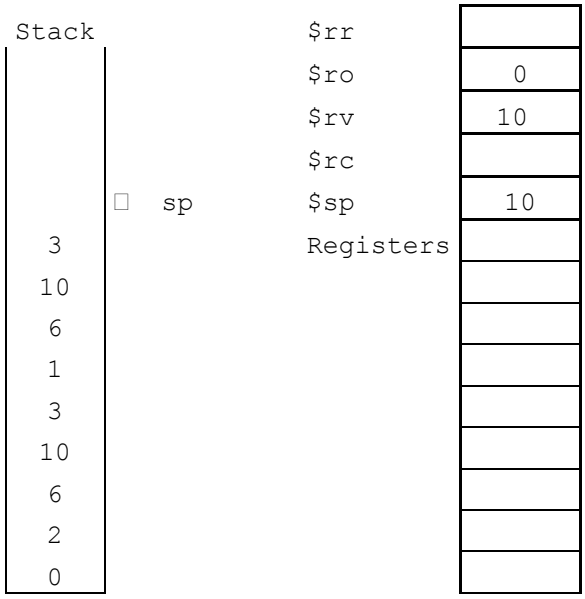
The value of b is copied to the registers, decremented and the new value is copied back to b's position in the stack.

```
Add    $ro    $ro    2
Copy   sp-2   $r1
Dec    $r1    1
Copy   $r1    sp-2
Setv   sp     10
Inc    $sp
Inc    $ro
```



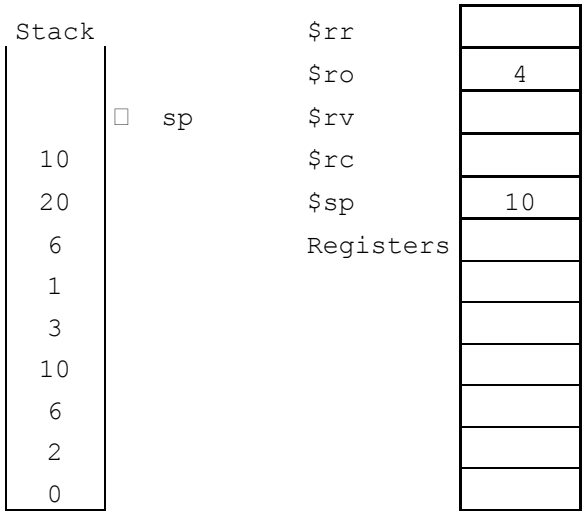
This time the condition is met and the recursive call can terminate. The value of d (10) is copied to the return value register and a's parameters are popped from the stack.

```
Setv   $rv   sp-1
Sub    $sp   $sp   3
```



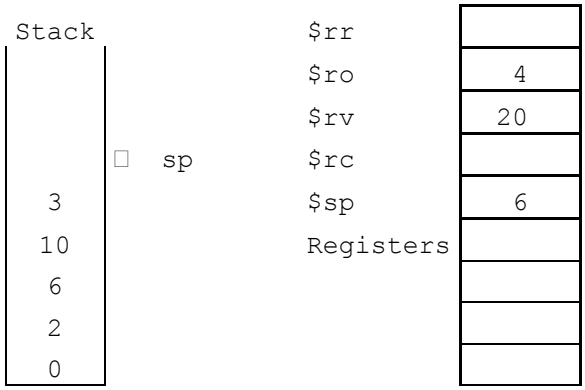
The caller procedure offset is restored. The return value is pushed onto the stack and d is set to be equal d + the return value.

```
Copy   sp-1  $ro
Dec    $sp
Setv   sp    $rv
Inc    $sp
Inc    $ro
Copy   sp-2 $r2
Copy   sp-1 $r1
Add    $r3   $r2   $r1
Copy   $r3  sp-2
```



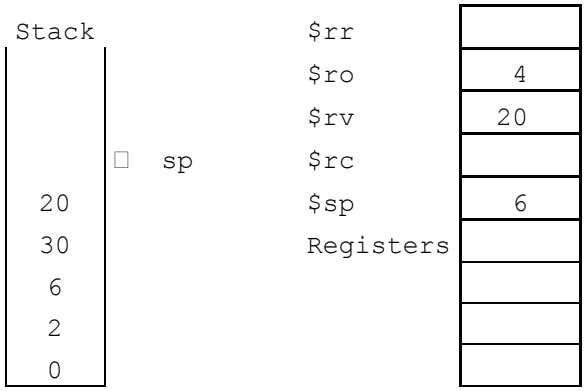
The value of d (20) is copied to the return value register and a's parameters are removed from the stack.

```
Setv   $rv   sp-2
Sub    $sp   $sp   4
```



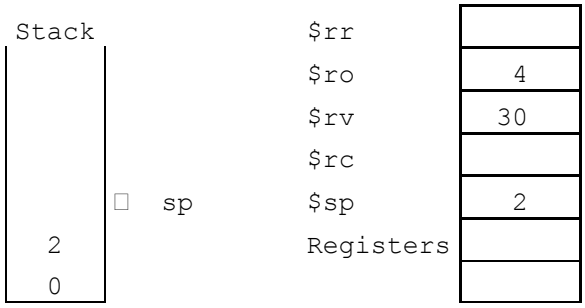
The caller procedure offset is restored. The return value is pushed onto the stack and d is set to be equal d + the return value.

```
Copy   sp-1   $ro
Dec    $sp
Setv   sp     $rv
Inc    $sp
Inc    $ro
Copy   $r1    sp-2
Copy   $r1    sp-1
Add    $r3    $r2    $r1
Copy   sp-2   $r1
```



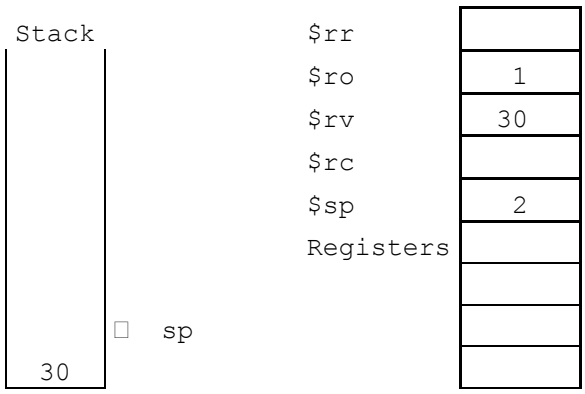
The value of d (20) is copied to the return value register and a's parameters are removed from the stack.

```
Setv   $rv    sp-2
Sub    $sp    $sp    4
```



Finally the recursive call has returned to the main procedure and x is assigned to the return value of the recursive call.

```
Copy   sp-1   $ro
Dec    $sp
Setv   sp     $rv
Inc    $sp
Inc    $ro
```



16. Code optimization

The current implementation of code generation produces statement by statement target code. Such code will often produce redundant instructions and far from optimal code constructs. It can often be optimized significantly by applying certain transformations to the target code after or during code generation.

Redundant load and stores

When generating code for statements like

```
a = b + c;
d = a + e;
```

The following code would be generated

```
(1)  MOV   b,          R0
(2)  ADD   c,          R0
(3)  MOV   R0,         a
(4)  MOV   a,          R0
(5)  ADD   e,          R0
(6)  MOV   R0,         d
```

In this case instruction (4) are redundant since the value for a is already in register r0. If a is not used further in the code, instruction (3) could be removed to.

Compile time expressions
At times people will write expressions that can be evaluated at compile time. This is often done when using constants and people want to make code easily read because a certain calculation makes more sense than just a value. A statement like

```
a = 5 + 2;
```

Will generate the following

```
(1)  MOV   5           R0
(2)  ADD   2           R0
(3)  MOV   R0          a
```

In this case, computing the expression during runtime is a waste, and the code could be optimised to.

```
(1)  MOV   7           a
```

Unreachable code
Unreachable code can be generated when conditionals, will always evaluate to the same. Such code might look like.

```
final boolean debug = false;

if (debug) {
    //  print debug info
}
```

The translated code may be translated to

```
(1)  MOV   0           debug
(2)  MOV   debug       CX
(3)  JZ    L1          // jump if register CX = zero
(4)  print debug info
(5)  L1:
```

In such a case everything except (1) could be discarded since instruction (3) will always jump to (5).

If additional time had been available in the project, the presented optimization strategies could have been implemented. This could be done by making a final scan of the target file after code generation. The scan would search for certain instruction constructs. If found they would be replaced with optimized constructs. Such optimization strategy is known as peephole optimization.

17. Test

For test purposes there are constructed a number of input files for the compiler. The files contain source code with errors, to which the compiler responds in a given way. For each error there is a comment describing, how the compiler responds. The test files can be seen in Appendix 17. The test files are listed below and are located in the directory named test:

- final.txt
- identifier.txt
- if.txt
- proc_func.txt

- rcx_function.txt
- repeat.txt
- scope_symboltable.txt
- source.txt (working example)
- variable.txt
- while.txt

During the test of the symbol table and the semantic check some minor bugs was discovered and corrected. These bugs were concerning wrong line numbers in some of the error messages.

Some of the semantic error messages are cascading errors, caused by another semantic error. For example using an undeclared identifier as right side of an assignment will display two error messages, one error message to inform about the undeclared identifier and one because the types do not match each other.

Some errors will make the ErrorHandler go into an infinite loop. One of these errors are unbalanced braces/brackets, which makes the error handler attempt to recover by going backwards to find the missing brace/bracket. Whenever such a loop is discovered, the compiler is stopped by force.

18. Conclusion

A high-level procedural language was defined for a next generation RCX-unit. The language supports global and local variables, constants and recursive procedures.

The language specification was fully implemented, but had a few flaws. One of them was the case of the functions keyword and another case was the limitation of only allowing a return statement in the end of a procedure.

For the language, a compiler was developed, with error handling that detects lexical, syntactic and semantic errors, allowing multiple error messages at a time, which will aid a programmer in debugging a program. The language uses a lexical scope without nested procedures. For some of the errors, error productions were made. The error productions made it easier to throw errors, but could only be implemented for productions not resulting in reduce/reduce or shift/reduce conflicts. All the intended semantic checks were also fully implemented.

The compiler was implemented with SableCC, a compiler compiler tool. SableCC uses an object oriented framework and supplies a lexer and a parser from the language specification. This makes it easy to make alterations when discovering them, without having to rewrite code. A symbol table was not integrated in SableCC so we designed and implemented one.

The target code were midway changed from hex-code to a self-defined assembler language, inspired by NQC's intermediate code and the 8086/8088 assembler language [L&G86]. This change was due to recursive procedures, which was not supported by the RCX-unit memory allocation. This means that the self-defined assembler also supports a stack implementation, made by ourselves.

Code generation was not completed. The recursive procedures were implemented, but since some of the essential parts of the code generation were not implemented,

we are not able to generate a functioning program, even if we had a modified RCX-unit.

We did consider some code optimization, but did not implement any of it, because we did not prioritize it as much, as the other phases.

Having been through all the above phases, we have understood the theory and proven our ability to use it to some extend in practice.

19. Appendices

19.1. Appendix A - References

Books

[ASU88] Compiler Principles, Techniques, and Tools
Addison Wesley Longman 1988
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

[APP97] Modern Compiler Implementation in Java
Cambridge University Press 1997
Andrew W. Appel

[FOR96] PC Principper
Teknisk Forlag 1996
Gunnar Forst

[L&G86] Microcomputer systems: The 8086/8088 family
Prentice-Hall 1986
Yu-Cheng Liu
Glenn A. Gibson

Web

[GAG98] SableCC, an object-oriented compiler framework
McGill University 1998
Étienne Gagnon
http://www.sablecc.org/thesis.pdf

[WSM00] JavaCC documentation
WebGain and Sun Microsystems 2000
http://www.webgain.com/products/java_cc/documentation.html

[BAU02] Not Quite C documentation
David Baum
http://www.enteract.com/~dbaum/nqc

[OVE00] RCX Command Center
Mark Overmars
http://www.cs.uu.nl/people/markov/lego/rcxcc/

[HAN02] Bricx Command Center
John Hansen
http://members.aol.com/johnbinder/bricxcc.htm

19.2. Appendix B - Directory structure and files

Organisation of the compiler source code.

The compiler source code is located in the following directories:

- grammar: the specification file for the SableCC framework.
- legoErrorProduction: the tree walker responsible for prompting error messages for the error productions.
- legoLexer: the lexical analyser.
- legoParser: the parser with error handling.
- legoSablecc: the files generated by the SableCC framework. This directory is divided into five subdirectories:
    - analysis: the tree walkers to be extended by the customized tree walkers.
    - errorrecovery:
    - lexer: the lexical analyser to be extended by the customized lexical analyser.
    - node: Java files representing the different terminals and non-terminals.
    - parser: the parser to be extended by the customized parser.
- legoSemantic: the tree walker responsible for performing semantic check.
- legoSymbolTable: the symbol table.
- legoTranslation: the tree walker for translation.
- test: different input files to the compiler for different types of tests.

The file Compiler.java is the source file binding the different parts of the compiler together.

19.3. Appendix D - Grammar tree

Error productions are not shown.

19.4. Appendix E – Test files

The test files are located in the directory named test.

final.txt

```
/*
Finals are declared and tested for syntax end semantic correctness.
*/

final int aaa = 0;
final boolean bbb = true;

// syntax error
final void ccc = 0; // illegal type void
final void ccc = true; // illegal type void

// semantic error
final int ddd = true; // boolean cannot be applied to integer
final boolean eee = 0; // integer cannot be applied to boolean

// error production
final int fff = 0 // missing ;
final int ggg; // undeclared constant
final int ggg := 4; // Pascal assignment

functions

void main()
{
   aaa = 0; // semantic error/symbol table error, aaa is a constant
   bbb = true; // semantic error/symbol table error, aaa is a constant
}
```

rcx_function.txt

```
/*
Use of the rcx function is demonstrated.
*/

functions

void main()
{
   wait(5);

   Wait(5); // Wait should be wait
   wait(); // missing parameter

   playTone(5, 5);

   playtone(5, 5); // playtone should be playTone
   playTone(5); // missing parameter
   playTone(); // missing parameters

   goRev(LEFT);
   goRev(RIGHT);
   goRev(BOTH);

   gorev(LEFT); // gorev should be goRev
   goRev(5); // 5 should be an engine selector
   goRev(); // missing engine selector

   goFwd(LEFT);
   goFwd(RIGHT);
   goFwd(BOTH);

   gofwd(LEFT); // gofwd should be goFwd
   goFwd(5); // 5 should be an engine selector
   goFwd(); // missing engine selector

   stop(LEFT);
   stop(RIGHT);
   stop(BOTH);

   Stop(LEFT); // Stop should be stop
   stop(5); // 5 should be an engine selector
   stop();  // missing engine selector

   setSpeed(LEFT, 5);
   setSpeed(RIGHT, 5);
   setSpeed(BOTH, 5);

   setspeed(LEFT, 5); // setspeed should be setSpeed
   setSpeed(LEFT); // missing parameter
   setSpeed(); // missing parameters
}
```

identifier.txt

```
/*
Tests identifiers for lexical and semantic corretness.
*/

int aaa = 0;
int 0bb = 0; // lexical error, identifier may not start with digit

functions

void 0xx()// lexical error, identifier may not start with digit
{
}

// Errorhandler goes into parse loop
```

if.txt

```
/*
The if control structure is tested for semantic corretness.
*/
boolean gretina1 = true;
boolean gretina2 = false;

boolean bool1 = true;
int int1 = 1;

boolean bool2 = false;
boolean bool3 = true;

functions

void main()
{
   if (true)
   {
   }

   // semantic error
   if (0) // the condition should be evaluated to type boolean
   {
   }

   // error production
   if (gretina1 = gretina2) // use of assigment operator instead of comparison operator
   {
   }

   if (bool1 == int1) // Types must be the same for operator "=="
   {
   }

   if (bool2 == bool3) // Correct
   {
   }

   if (int1 == int1) // Correct
   {
   }
}
```

source.txt

```
/*
Contains correct source code.
*/

functions

void main(){
		turnLeft(1, 2, 3);
		stop(LEFT);
		turnRight(9);
}

void turnLeft(int a, int b, int c){
		goFwd(LEFT);
		stop(LEFT);
}


void turnRight(int a){
		goFwd(LEFT);
		stop(LEFT);
}
```

proc_func.txt

```
/*
Procedures are declared and tested for lexical and semantic correctness.
*/

functions

void aaa()
{

}

void epFunc2(int ep1; boolean ep2) // wrong parameter declaration delimiter
{
   epFunc3(42; true); // wrong parameter delimiter
}

void epFunc3(int ep1, boolean ep2)
{

}


int epFunc4()
{
		return 0 // missing semicolon at return statement
}


int epFunc5()
{
		return 0;
}


boolean epFunc6(int ep1, boolean ep2)
{
		epFunc7(5,5); // Too many parameters
		epFunc7(true); // Incorrect type of parameter
		epFunc7(); // Too few parameters
		return 0; // Incorrect return type
}

void epFunc7(int ep3)
{
		return 0; // Incorrect return type
}



void epFunc8(int ep3, boolean ep3) // Names of parameters must be unique and not
redeclared
```

```
{
   epFunc9();
            epFunc10();
}

void epFunc9()
{
   xxx(iProc());
   xxx(bProc()); // bProc() returns boolean, xxx(int i) takes an integer as parameter
}

void xxx(int i)
{
}

int iProc()
{
   return 666;
}

boolean bProc()
{
   return true;
}
```

repeat.txt

```
/*
Use of the repeat control structure is demostrated and tested for lexical and semantic
corretness.
*/

functions

void main()
{
   int a=2;
   boolean b=true;

   repeat(0)
   {
   }

   repeat(true) // wrong type
   {
   }

   repeat(5*6+3)
   {
   }

   repeat(a)
   {
   }


   repeat(b) // wrong type
   {
   }

}
```

scope_symboltable.txt

```
/*
The symbol table is tested by trying to violate the rules of scope.
*/
int a = 100;
int b;

functions

void main()
{
   int x = 0;
   aaa();
}

void aaa()
{
   int y = x; // semantic error - 'x' is not defined in global scope or in scope of
procedure 'aaa'.
              // Would be correct if scope was dynamic.
   a = 5;     // Correct uses 'a' from gloabal scope
   b = true;  // Boolean is not correct type
   b = 5;     // Correct uses 'b' from gloabal scope

   int life = 42;

   while (true)
   {
      life = a;
   }
}
```

variable.txt

```
/*
Both assigned and unassigned variables are declared and tested for syntax end
semantic correctness.
*/

int aaa = 0;
boolean bbb = true;
int ccc;
boolean ddd;
int xxx;


// syntax error
void eee = 0; // illegal type void
void fff = true; // illegal type void


// semantic error
int ggg = true; // boolean cannot be applied to integer
boolean hhh = 0; // integer cannot be applied to boolean

// error production
int jjj // missing ;
int iii = 0 // missing ;
boolean kkk := true; // illegal assigment operator
int kkk := 0; // illegal assigment operator


functions

void main()
{

   aaa := 0; // error production, illegal assigment operator
   aaa := 0  // error production, missing semicolon
   mmm == 0; // semantic error must use = instead

   int aaa = 0; // NO ERROR, although identifier is also used in global scope

   mmm = 0; // semantic error/symbol table, variable not declared
   int yyy = xxx; // semantic error - xxx is not assigned

}
```

while.txt

```
/*
The if control structure is tested for semantic corretness.
*/
boolean gretina1 = true;
boolean gretina2 = false;

boolean bool1 = true;
int int1 = 1;

boolean bool2 = false;
boolean bool3 = true;

functions

void main()
{
   while (true) // correct
   {
   }

   // semantic error
   while (0) // the condition should be evaluated to type boolean
   {
   }

   // error production
   while (gretina1 = gretina2) // use of assigment operator instead of comparison
operator
   {
   }

   while (bool1 == int1) // Types must be the same for operator "=="
   {
   }

   while (bool2 == bool3) // Correct
   {
   }

   while (int1 == int1) // Correct
   {
   }
}
```