Lab Instructions for Compiler Theory

Alexandre David

Contents

Chapter 1

Introduction

1.1 The Project Assignment

The project assignment is to construct a compiler for a small programming language called AFL using the tools *Flex* and *Bison*. Appendix C contains a definition of AFL. The compiler should generate code for an absract machine called AFM which is described in Appendix D. The compiler will be constructed in three steps each of which is reported separately. Table 1.1 contains a brief description of the three steps and their due dates. Note that you need to complete each step to be able to do the next one.

1.2 Course Information

You may find information about the assignment, results etc. on WWW under the following URL:

`http://www.docs.uu.se/~adavid/courses/compiler/vt2002.html`

	Due date	**Description**
Step 1	20/2	Use the tool Flex to construct a lexer
Step 2	26/2	Use the tool Bison to construct a parser
Step 3	31/3	Write code for generating AFM-code

Table 1.1: The three steps of the project assiggnment.

If you find it too taxing to type this address, try locating the appropriate page under my home page

`http://www.docs.uu.se/~adavid`

Questions concerning the project assignment will be answered by Alexandre David, room 1237, email: `adavid@docs.uu.se`. Solutions to project assignments should be left in my mailbox on the 4th floor of house 1.

1.3 Reporting the Result

When reporting the results, you are required to turn in the following:

Title page The title page should contain the name(s) of whoever did the assignment. It should also contain the location of an *executable* program which I can run.

The title page does not necessarily have to be an entire page, but it is important that it contains the name (to let me know who wrote the program) and the path to an executable program (so I can test your program). Assignments where any of this information is missing will *not* be corrected.

Presentation This is a description of the work you have done. This includes a description of the problem you were given.

If you had any problems or you have any comments, this is where you write it.

Solution This is usually the code you wrote to solve the assignment. For each assignment there is a special section telling what you *have* to turn in.

Note! Any written code should be clear, concise and adequately documented (see Section 1.4)! If there is something in the code that I cannot understand, you will have to complete it by rewriting the code.

Demonstration As you are writing programs, you have to demonstrate that they are correct. You do this by providing an input and show the corresponding output generated.

If you write your own examples, please demonstrate each feature separately. It is usually very hard to see if something is correct when given a huge test run that you have to sort through.

Hint! If you have the example input in file `input.ex` and the resulting output in file `output.ex` you can get both files printed side by side on a paper using the command

```
a2ps -A fill -Pprinter input.ex output.ex
```

1.4 Documenting the Code

In general you can say that a program can be documented implicitly and explicitly. Implicit documentation is

- To use good names for variables and functions (i.e. `max_value` instead of `mxvlu`, etc.)
- Good indentation and grouping that illustrates what code belongs where and other stylistic ideas that make the code easier to read and understand. An example that have caused many headaches is

  ```
  if (a == b)
     a++; b++;
  ```

  If we rewrite it properly, it is easy see what the code actually does.

  ```
  if (a == b)
     a++;
  b++;
  ```

The explicit documentation consists of a collection of descriptions of the parts of the program. There are a few cases where any program should be explicitly documented.

1. Each file should have a header containing
   (a) The *name and path* of the file
   (b) The *author*
   (c) A description the intended *purpose* of the file

This is an example of a file-header:

```
/*****************************************************************
 *
 * File:       lexer.l
 *
 * Path:       ~ingen/course/comp/lab-1/
 *
 * Author:     Inge Naning
 *
 * Date:       991231
 *
 * Purpose:    Flex specification of a lexer for AFL
 *
 *****************************************************************/
```

2. A header for each function, even if the function is small. The header should contain

   (a) A description of what the function does.

   (b) A description of how the arguments should be used and what values are allowed.

   (c) What value the function returns, or a note that it does not return anything.

   (d) Any additional information that might be necessary to know when using the function.

   An example of a function description is:

```
/*
 * int gcd(int a, int b)
 *    This function computes the greatest common divisor of a and b.
 * Note:
 *    It is assumed that both a and b are positive. An undefined
 *    result will be returned if a or b are negative.
 */
```

3. A comment for each block of code that does something distinct, e.g.

```
/* move tmp to point to the first non-empty string in ‘list’ */
for (tmp = list.first ; tmp ; tmp = tmp->next)
  if (strlen(tmp->value) > 0)
    break;
```

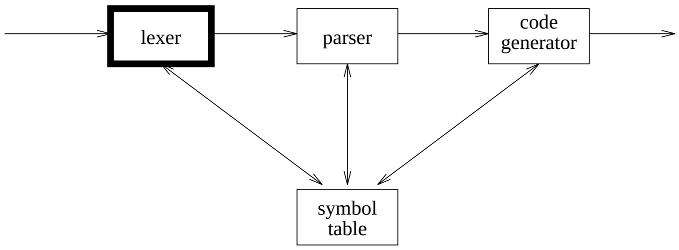
4. Comment if something is not clear from the code. Example:

```
/* Either a == 0 or b == 0 but not both. */
if ((a == 0) + (b == 0) == 1)
  printf("Hello\n");
```

   Avoid writing an excessive number of comments of this type since they have a tendency to clutter up the code. Especially, do *not* write comments inside the code, put them last on a line or on a separate line.

Chapter 2

Step 1: Writing the lexer



2.1 Preparations

In preparation for this lab you should read the definition of AFL in Appendix C. Especially study the definition of tokens and comments.

You should also be familiar with regular expressions, how to write them and what they mean. Chapter 3 of *Compilers: Principles, Techniques, and Tools* contains information about lexical analysers. Especially read 3.3 – 3.5 and maybe 3.6 – 3.8 to familiarise yourself with how a lexical analyser works. It is also important that you know how to write Flex specifications and how to run Flex.

2.2 Assignment

The first assignment is divided into two parts, each of which is actually a small assignment. The first part is to write a filter that recognises some patterns in a text. The second part is to write the lexer for your compiler.

Part 1: Filter

Use flex to generate a program “filter”, that reads a text from standard input and writes a filtered version of the text to standard output. The filtered version of the text should be the same as the original text *except* in the following cases:

1. The words “COBOL” and “FORTRAN” (in any case of letters) should be replaced by the string “CENSORED”.
2. Integers should be replaced by the string “INTEGER”.
3. After any word containing two or more (lowercase) “c”, separated only by lowercase letters, the string “`<ta-ta>`” should follow.

Make sure that *all* characters read from standard input are written to standard output (unless they form COBOL or FORTRAN, of course!).

Example: If the program is fed the following text

```
 Throughout the 1950’s, compilers were considered
notoriously difficult programs to write.  The first
Fortran compiler, for example, took 18 staff-years to
implement.  We have since discovered systematic
techniques for handling many of the important tasks that
occur during compilation.
```

the output should be

```
 Throughout the INTEGER’s, compilers were considered
notoriously difficult programs to write.  The first
CENSORED compiler, for example, took INTEGER staff-years
to implement.  We have since discovered systematic
techniques for handling many of the important tasks that
occur <ta-ta> during compilation.
```

Part 2: Lexer for AFL

This part of the assignment consists of writing the lexer for your AFL compiler. The lexer should:

1. Recognise all lexemes and return the *symbolic name* (see Table C.1) for the corresponding token.

2. Strip away comments, i.e., the lexer should recognise comments but not do anything when one is found.

3. Keep track of the line number. On page 41 you have a description of the variable `Line_number`, which is the line number counter.

4. Return the lexical values of identifiers and numbers as strings and integers respectively.

5. If something that does not match any of the lexemes is read, an error message should be printed.

The files `y.tab.h` and `other.c`

To do this part of the lab you need a file `y.tab.h` containing definitions of tokens and a definition of `yylval`. The variable `yylval` can contain either an integer or a string. To give `yylval` an integer value assign the value to `yylval.integer`, to give it a string value assign the value to `yylval.string`. In the next lab you will create your own `y.tab.h`, but for this lab you can copy
`/stud/docs/kurs/kompteo/lab-1/y.tab.h`.

In the same directory you will also find a file `other.c` containing a `main`-routine that you should use for this lab. If you compile `other.c` together with your `lex.yy.c` as described on page 24 you will have an executable program that prints the tokens that are found.

This scanner will later be used by the parser, which means that it has to recognise all tokens that are in the language. The tokens should then be returned to the parser. The default action for the lexer is to echo unrecognised characters to the terminal, which is not a desired behaviour for a compiler. If anything unrecognisable is seen, the lexer should give an error message. To print error messages to the terminal you should use the function `ceprintf`, which is described in Appendix B (page 41).

2.3 Report

For both part 1 and part 2 you should supply:

- the Flex specification, and

- demonstration of correct behaviour. Each demonstration should consist of an example input (a text in part 1 and a program in part 2) and the resulting output.

Don't forget to submit the paths of *two* executable programs, one for each part of the assignment!

2.4 Files

`/stud/docs/kurs/kompteo/lab-1/sample[1-2].txt` :
Texts to test the filter

`/stud/docs/kurs/kompteo/lab-1/other.c` :
`main`-routine

`/stud/docs/kurs/kompteo/lab-1/y.tab.h` :
Definitions of tokens

`/stud/docs/kurs/kompteo/lab-1/test.afl` :
Test program

`/stud/docs/kurs/kompteo/lab-1/test.run` :
Correct output from running the test program

2.5 Common Errors

Following is a few of the more common errors that you should avoid.

- Writing an excessive number of brackets ([and]) in the regular expressions. Note that `[a]*` and `a*` are equivalent regular expressions.
- Not taking care of whitespace (space and tab).
- Forgetting to copy `yytext` (see Section A.1.3).
- Forgetting to write a "catch-all" rule that takes care of illegal characters.
- Using incorrect definitions of tokens. Please consult Appendix C to get the correct definitions.

int **atoi**(char* str)

Description:
This function returns the integer value represented by the string pointed to by `str`. The string is scanned up to the first character that is not a digit. Leading white-space characters are ignored.

int **strdup**(char* str)

Description:
This function returns a pointer to a new string which is a duplicate of the string pointed to by `str`. The space for the new string is obtained using `malloc(3V)`. If the new string cannot be created, a NULL pointer is returned.

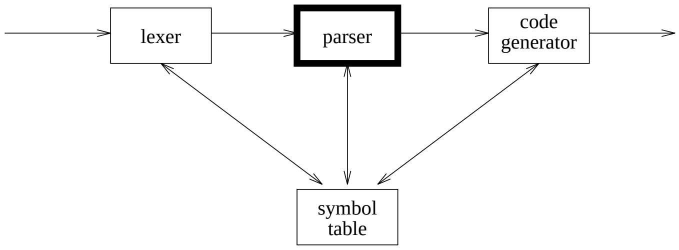
Figure 2.1: Useful functions.

Some problems that might occur when using Flex is:

- Sometimes Flex does not expand shorthands correctly, which will generate very strange errors ("unrecognised rule"). Correct this by hand-expanding the shorthands in the rules.
- Flex might be *very* picky about empty lines and comments in the definitions and rules section. Avoid empty lines and comments as much as possible in those sections.

Chapter 3

Step 2: Writing the parser



3.1 Preparations

In preparation for this lab you should read the definition of AFL in appendix C. Especially study the grammatic definition of AFL, which involves the grammar, the precedence of operators and other syntactic constructions.

Make sure you know how to write Bison specifications and how to run Bison. Read the chapters on Yacc (section 4.9, page 257–266) and translation schemes (section 5.4, page 297–301) in *Compilers: Principles, Techniques, and Tools*.

You should also read about the symbol table functions in Appendix B. Make sure you understand how to use markers.

3.2 Assignment

The assignment for this lab is to write grammar and code to check that a program is syntactically and semantically correct.

Remove ambiguities from the grammar in Table C.3. You can remove the ambiguities by:

1. rewriting the grammar

2. using Bison's `%left`, `%right` and `%nonassoc` commands to declare the precendece of the grammar rules.

Add semantic actions that stores identifiers in the symbol table to the grammar. You can find a complete description of the symbol table in Appendix B.

In addition you will have to write a `main` that prints the symbol table after the parsing succeeded.You can find a template for a `main` at page 26. To print the symbols in the symbol table, use the function `print_symbols` that is in the symbol table library.

You are *not* allowed to use global variables, use the $-variables instead.

Example Programs

To test that your parser works correctly there are two AFL programs in `/stud/docs/kurs/kompteo/lab-2` by the names `badafl1` and `badafl2`. The parser should be able to parse these AFL programs without causing a "segmentation fault" or "bus error". It should also print correct error messages.

3.3 Report

For this lab you should supply the following:

1. The changes you made to remove ambiguities. (Describe them in your report.)

2. The Bison specification with semantic actions that inserts symbols into the symbol table and reports errors.

3. A demonstration that you code works. Each demonstration should consist of:

   (a) an example input (a program) and

   (b) the resulting output (error messages).

Example: The AFL program `badafl1` looks as

```
apples(n) := if n = 0 then 1 else n * apples(n-1);
bananas := 3;
oranges(apples,a,a):=1 + a(1);
bananas := 0;
?
apples(4,2) + zucchini(4) + broccoli  + apples
```

and the output from a parser should be something like

```
line 3: Invalid parameter name, apples is already defined
line 3: Invalid parameter name, a is already defined
line 3: a is not a function
line 4: Invalid function name, bananas is already defined
line 6: Wrong number of actual parameters for function apples
line 6: Undefined function zucchini
line 6: Undefined symbol broccoli
line 7: Wrong number of actual parameters for function apples

Symbol table contains:

bananas : function with 0 parameters
oranges : function with 3 parameters
bananas : function with 0 parameters
apples : function with 1 parameters
```

3.4 Files

`lib{solaris,sunos}-kompteo.a` :
The libraries containing symbol table functions, see appendix B.

`/stud/docs/kurs/kompteo/lab-2/badafl[1-2]` :
Faulty AFL programs

`/stud/docs/kurs/kompteo/bin/compile` :
Helpful script, see section B.1.

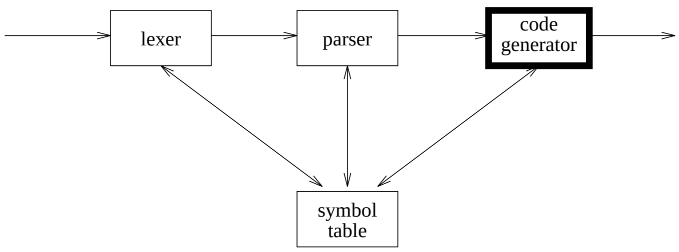
3.5 Common errors

Following is a list of a few of the more common errors that you should avoid.

- Assigning values to the `$`n variables.
- Not assigning a value to `$$` in a rule which you use the value of. You can see an example of that on page 33.
- Assigning a value to `$$` in a rule which you do not use the value of.
- Using global variables. Don't do this, you don't have to.
- Assigning values to `$`n-variable. You should *never* assign a value to a numbered variable like `$1` or `$2`. This does not include the `$$` variable.
- Reading or updating the value of `$$`. You should *only* assign values to `$$`, never read. Not even implicitly, as in `$$++` or `$$ *= 4`.
- Forgetting to include the file `kompteo.h`, as described on page 35.
- Using the `other.c` file from Lab 1. You should write you own at this stage.

Chapter 4

Step 3: Writing the code generator



4.1 Preparations

In preparation for this lab you should study the definition of AFM (Abstract Functional Machine) in appendix D. *Study the entire appendix carefully!*

Exercise

To find out what code is to be generated in this lab you should write all rules on how to convert a AFL construction into AFM code. Use the syntax-directed definition from *Compilers: Principles, Techniques, and Tools*. As an example the rules for the addition rule and the constant number rule follows.

PRODUCTION	SEMANTIC RULES
$E \to E_1 + E_2$	$E.code := E_1.code \parallel E_2.code \parallel gen(\texttt{ADD})$
$E \to \textbf{num}$	$E.code := gen(\texttt{PUSH}\ \textbf{num}.value)$

The *code* attribute contains the generated code and the *value* attribute contains the value of a numerical token.

4.2 Assignment

The assignment for this lab is to extend the specification from lab 2 with C-code that implements the code generation of your compiler. This means that you have to write semantic actions that generates AFM-code given some AFL-code.

When this lab is finished you should have a complete compiler for AFL. This means that the AFM-code that the compiler generates should be executable by the AFM simulator (see page 51).

4.2.1 Advice and hints

- Write the generated code to a file. The simplest way to do this is using print statements in the semantic actions, for example:

  ```
  expression : NUMBER { printf("PUSH %d\n", $1); }
  ```

  Assume that your Bison generated parser is in the file `a.out`, the AFL program is in the file `file.afl`. You can then write this generated code into the file `file.afm` by:

  ```
  ./a.out <file.afl >file.afm
  ```

- Don't try to solve the whole lab at once, instead, divide the problem into smaller parts. For instance one could start with arithmetic expressions (Expression), so you can generate code for the following AFL program:

  ```
  ?  3*12+9
  ```

  Continue with function definitions and finally conditional expressions (`if-then-else`).

- Use the exercise from the Preparations section to help you decide what code to generate for each language construction.

You are **not** allowed to use global variables in this lab (you will not need any global variables). There are two cases when you may be *tempted*:

Generating Labels Code generated from conditional expression includes conditional jumps. The compiler must generate symbolic labels for these jump statements. In order to do this you can use the function `newlabel` for which you can find a description in appendix B. This function returns a new label every time it is called. Using this function you can generate a new label as follows:

```
{ printf("GOTRUE %s", newlabel("L"));}
```

This would generate `GOTRUE L0`. If executed again, it would generate `GOTRUE L1`. The following example illustrates how to generate the value of a previously generated label[1].

Example. Assume `get_current_time` is a function which returns the current time and `NUMBER`, `PLUS` and `SEMICOLON` are terminals. The following is a translation rule in a Bison specification:

```
statement : the_time
              { printf("start time: %d", $1); }
            NUMBER  PLUS  NUMBER the_time
              { printf("end time: %d", $6); }
            SEMICOLON
              { printf("total time = %d", $6 - $1 );  }
          ;

the_time  : /* empty */
              { $$ = get_current_time(); }
          ;
```

In this example the value of the rule `the_time` is set to the current time. In this manner, we can reference the value of `the_time` later in the rule `statement` as `$1` and `$6` respectively.

For this example to work correctly we have to declare the type of the `the_time` rule. We do this by first defining a new type `time` using the `%union` command (see page 28) and then declaring the type of `the_time` with the command

```
%type <time> the_time
```

Pushing parameter values On page 47 is a description of the format of an activation record for AFM. "the last parameter is always at address

[1] Don't try generating it from `newlabel`, `newlabel` gives you a new number every time it is called.

$fp + 2$, the next to last at address $fp + 3$, etc.". In order to calculate where each parameter is on the stack, you need to know:

- the parameter number
- the total number of parameters in the activation record

These values can be obtained from the symbol table, using the functions `get_paramno`, `get_last`, and `get_no_param` (assuming that you have inserted them there, of course).

4.3 Report

For this lab you should include the following:

1. The Flex specification.
2. The Bison specification, enriched with semantic actions that generates AFM-code.
3. Demonstrations that you compiler works properly. Each demonstration should consist of:
   (a) the AFL-code you send to your compiler,
   (b) the resulting AFM-code, and
   (c) the result from executing the AFM-code.

   You should compile and execute several (2–3) small AFL programs to demonstrates different aspects of the compiler. You can find some test programs under `/stud/docs/kurs/kompteo/lab-3`, but please write some small test programs yourself.

4.4 Files

`/stud/docs/kurs/kompteo/bin/afm` :
: AFM simulator

`/stud/docs/kurs/kompteo/lab-3/test[1-4].afl` :
: Test Programs

`/stud/docs/kurs/kompteo/lab-3/test1.afm` :
: AFM code example

4.5 Common errors

Following are a few of the more common errors that you should avoid.

Not specifying the associativity and/or precedence of operators. This can be avoided if you read appendix C carefully. If you have specified the associativity incorrectly for substraction (which is non-commutative) you will get strange results when compiling an expression involving several subsequent substractions.

Not computing the value to `PUSHIND` correctly. You can catch this error by writing several small functions with different number of arguments and "hand-running" the resulting AFM-code.

Appendix A

Flex & Bison

A.1 Flex

The first phase of a compiler is the *lexical analysis*. Its main task is to read input characters and produce a sequence of tokens that the parser later uses for syntax analysis. Writing a lexical analyser (or *lexer* for brevity) is a very tedious task and is better done by a *lexical analyser generator*. Flex is such a tool.

Flex takes a description of tokens to match and actions to perform when the tokens are matched. A description of Lex can be found in section 3.5 of *Compilers: Principles, Techniques, and Tools* and everything said there applies to Flex. In figure A.1 you can find a complete list of the regular expressions available when writing a Flex specification. You can find the same table on page 148 in the book mentioned above.

A.1.1 A Flex Specification

A Flex specification (Figure A.2) consists of three sections. The sections are separated by a line containing `%%`.

definition section The first section is called the *definition section*. In this section we may define *shorthands* for regular expressions. Between the symbols `%{` and `%}`, we may write C-code that is inserted literally in the beginning of the generated code.

rules section The second section is called the *rules section*. Each rule in this section has the form

EXPRESSION	MATCHES	EXAMPLE
c	a non-operator character c	`a`
$\backslash c$	character c literally	`\*`
$"s"$	string s literally	`"**"`
.	any character but newline	`a.*b`
^	beginning of a line	`^abc`
$	end of a line	`abc$`
$[s]$	any character in s	`[abc]`
$[\hat{}s]$	any character not in s	`[^abc]`
$r*$	zero or more r's	`a*`
$r+$	one or more r's	`a+`
$r?$	zero or one r's	`a?`
$r\{m,n\}$	m to n occurrences of r	`a{1,5}`
$r_1\,r_2$	r_1 followed by r_2	`a*b+`
$r_1 \vert r_2$	r_1 or r_2	`a*|b+`
(r)	just r (used for grouping)	`(a*|b+)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

c stands for any single character
r stands for a regular expression
s stands for a string of characters

Figure A.1: Flex regular expressions

regexp { *action* }

were *regexp* is a regular expression and *action* is a piece of C-code that is executed whenever the lexer finds something matching *regexp*.

user subroutines section The third section is optional and is called the *user subroutines section*. We will not use this section in the course.

When Flex generates a lexer it will build code that recognises the regular expressions and insert the actions at proper places. Everything is written as a very large subroutine called `yylex()`. The parser will call `yylex()` each time it wants a new token and it expects `yylex()` to return a *token id*. A token id is nothing but an integer representing the token, e.g. the token `IDENT` may have been assigned the token id 257. By default `yylex()` does *not* return a token id, which means that *you* have to write `return`-statements inside your actions if you want `yylex()` to return a token id. The token id's are stored in the file `y.tab.h` that is generated by Bison (see 29).

```
⟨Definitions⟩
    %{
      C-code
    %}

    shorthands for regular expressions

%%

⟨Rules⟩
    pairs of regular expressions and
    C-code

%%

⟨User subroutines⟩
    C-code, e.g main-program
```

Figure A.2: A Flex Specification

A.1.2 Running Flex

To construct an executable program using Flex there are a number of steps that has to be performed. In Figure A.3 you can see an illustration of the example below giving the order that the different compilers are called (the shaded boxes are files that you have written, the unshaded boxes are files that were generated by one of the programs).

Assume that you have written a Flex specification for your lexer into the file `lexer.l`, the support code into the file `other.c`, and all definitions of tokens are in the file `y.tab.h` (this last file is generated by Bison, but more about that in the next section).

To be able to use the definitions of all tokens you have to include `y.tab.h` in the beginning of `lexer.l`, this you do by writing the following lines *at the beginning of* `lexer.l`

```
%{
#include "y.tab.h"
%}
```

Flex will copy the part between `%{` and `%}` literally into `lex.yy.c`.

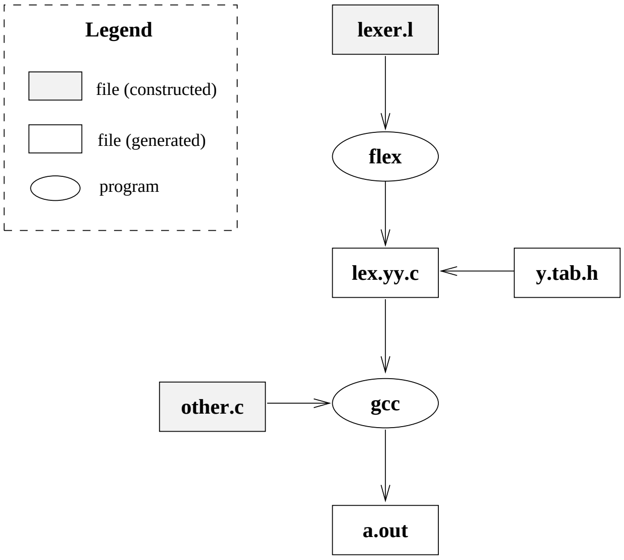


Figure A.3: Running Flex

Note! If you are using the *Library of Useful Functions* from appendix B, you have to include `kompteo.h` here!

1. Using the command

   ```
   flex lexer.l
   ```

   you call `flex` which will produce a file `lex.yy.c` containing C-code for a complete lexer.

2. You now have to compile and link all C-files together using the C-compiler. Also, there are a number of functions that Flex needs to work properly (`yyerror`, `yywrap`, `main`, ...). These functions are supplied by giving the switch `-lfl` to the C-compiler. The full command to compile the lexer is then:

   ```
   gcc lex.yy.c other.c -lfl
   ```

3. You now have an executable program in the file `a.out` which reads characters from standard input. If you want to execute the program and give it input from a file (`fact.afl`, in this case), you type the command:

   ```
   ./a.out < fact.afl
   ```

A.1.3 The global variables `yytext` and `yyleng`

When Flex has found a match it will execute the action associated with the regular expression that matched. Inside this action the global variable `yytext` will contain the string that matched, and `yyleng` will contain the length of the matched string.

Example: Assume that we have the following Flex rule:

```
[ab]*    { printf("%s", yytext); }
```

and that the input to the lexer is "`aabaaba`", when executing `printf`, `yytext` will be `"aabaaba"` and `yyleng` will be 7.

Note! If you want to save the string (for instance in a symbol table) you can *not* save `yytext`. The reason for this is that it points to an area that will be reused for the next match, which means that your string will be overwritten. What you should do to avoid this is to copy the string using the function `strdup` (**str**ing **dup**licate), which is described on page 9.

A.1.4 The global variable `yylval`

Every token may also have a *lexical value*. To send these values to a parser there exists a global variable `yylval` in which you store the lexical value you wish to return. The types of values you can return is declared using the `%union` command in the definition section (see section A.2.3).

Example: Assume that you have declared a `%union` that contains a float named `real` and an integer named `integer`. The following Flex rule will then, upon seeing a sequence of digits, return the token `CONST` with the integer's numerical value as lexical value.

```
[0-9]+    {
              yylval.integer = atoi(yytext);
              return CONST;
          }
```

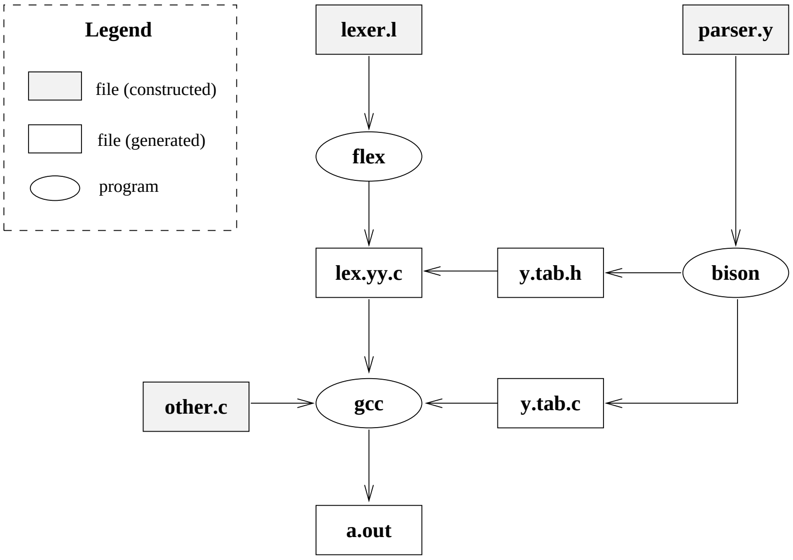


Figure A.4: Running Bison and Flex together

A.2 Bison

The second phase of a compiler is *syntax analysis*. The main task of this phase is to read the stream of tokens and verify that it can be generated by the grammar, i.e. that it is syntactically correct. Today, most modern languages are designed to be in a class of languages called LR(1). For this class there are a number of very efficient parsing methods. Unfortunately these methods usually require the construction of complicated tables — and code to handle the tables. You can compute these tables manually (see chapter 4 in *Compilers: Principles, Techniques, and Tools*), or you can use the tool *Bison*.

Given a specification containing a grammar and some other information Bison will generate a program that recognises the language specified by the rules in the specification. In reality the program is one large function `yyparse` that will analyse some text and return 0 if the parsing failed and 1 if it succeeded. A simple `main` that calls `yyparse` might be:

```
int main() {
  yyparse();
}
```

This function is preferably put in the file `other.c` mentioned below. If you link the code with `lib(y)` (see page 28) you will automatically get the `main` above.

A.2.1 A Bison Specification

A Bison specification consists of three sections. The sections are separated by a line containing `%%`.

definition section The first section is called the *definition section*, similar to the first section in a Flex specification. This section contains definitions of tokens and types. It also contain definitions of precedence between operators and is also used to introduce C-specific code into the parser program.

rules section The second section is called the *rules section*, also similar to the second section is a Flex specification. A rule in the rules section appears as follows:

$$\textit{non-terminal} \; : \; \textit{item}_1 \; \textit{item}_2 \cdots \textit{item}_n \; ;$$

where *non-terminal* is a name and each of the items may be either a token, a non-terminal, or a semantic action.

tokens are normally written using upper-case letters. Everything that is declared to be a token using the `%token` directive is considered a token.

non-terminals are normally written using lower-case letters. Everything that is not declared to be a token is considered a non-terminal.

semantic actions are pieces of C-code written inside braces, an example of a semantic action is `{ printf("Hello world\n"); }`. These are used to perform error checks, inserting symbols in the symbol table, etc.

user subroutines section The third section is called the *user subroutines section*. We will not use this section in the course.

A.2.2 Running Bison

If we now want to construct a full compiler using Flex and Bison we have to add a few steps to those on page 24. In Figure A.4 you can see a diagram on running Flex and Bison together. As you can see, Bison generates *two* files (`y.tab.h` and `y.tab.c`) given a Bison specification in a file `parser.y`.

Assume that the file `other.c` contains support functions for your compiler, `lexer.l` contains a Flex specification for your lexer, `parser.y` contains a Bison specification for your parser, and you want to build a compiler from these. The steps are then:

1. From `parser.y`, generate the files `y.tab.h` and `y.tab.c` by calling `bison`.

   ```
   bison -d -v -y parser.y
   ```

   The `-d` switch makes Bison generate a `y.tab.h` and the `-v` switch makes Bison generate a `y.output`. The `y.output` file is very useful for debugging (see below). The file `y.tab.h` will contain definition of tokens (have a look at it) and `y.tab.c` will contain C-code to implement your parser (not something you want to take a look at, trust me).

2. Now generate your lexer as described in step 1 in the previous section. Note that Bison generated the file `y.tab.h` for you.

3. Compile and link the C files together using the C compiler. As in the previous section, there are a number of functions that Bison needs. You supply these functions by adding the `-ly` switch. The full command to compile and link the files is then

   ```
   gcc lex.yy.c y.tab.c other.c -ly -lfl
   ```

   Note that the order of the `-lfl` and `-ly` switches are important here.

   Note! When doing the assignment you should use the `compile` script described in appendix B. The script will in addition to supplying the switches above also link the correct version of the *Library of Useful Functions* with your program.

4. You now have an executable compiler in the file `a.out`. This can be called in the same manner as in step 3 in the previous section.

A.2.3 Types and the `%union` command

In Bison every rule and every token may have a type. To declare types for rules and tokens you use the `%union`, `%token`, and `%type` commands that are parts of Bison's syntax.

Example: Assume that you have the rules:

```
expr    : id               { $$ = $1; }
        | expr PLUS expr  { $$ = $1 + $3; }
        | expr TIMES expr { $$ = $1 * $3; }
        ;
id      : ID               { $$ = lookup($1); }
        ;
```

and you want the rules `id` and `expr` to return integers, and the token `ID` to return a string. We create these types by writing the following command in the *definition section* of the Bison specification:

```
%union {
  int integer;
  char *string;
}
```

This will create two types named `integer` and `string` that we can use to declare the types of rules and tokens. To declare the token `ID` to be of type `string` and the non-terminals `id` and `expr` to be of type `integer` we write the following two lines into the definition section.

```
%type  <integer> id expr
%token <string>  ID
```

A.2.4 The file `y.tab.h`

In the definition section of `parser.y` there are a number of lines of the form

```
%token IF THEN ELSE WHILE
```

which are declarations of tokens (possibly with types, but more about that later). Bison will then generate a token id for each token and store it in `y.tab.h`. Any types declared using the `%union` command will also be put into the file `y.tab.h`. The LALR(1) table, together with code to support it, is written into the file `y.tab.c`.

`y.tab.h` contains C-code and it is not very long, so I suggest that you have a look at it to see what is put there.

Tokens have to be declared! If you do not declare the tokens, you will get an error message from Bison.

A.2.5 The file `y.output`

The file `y.output` contains information about the state table that Bison generated. The information is presented in a manner that is easier to read than the real table inside `y.tab.c`.

In each state there is:

- a list of all production rules that the parser is trying to match, with an dot '`.`' to indicate how much of the rule that has been recognised,
- a list of actions that the parser will perform, depending on the next token. The special token '`$default`' denote the default token (i.e. any token not listen earlier), and
- a list of non-terminals that gives the state that should be used if a reduction to the non-terminal has been performed.

An example, assume that we have the following grammar (a very small one) in the form of a Bison specification.

```
elem  :  ID ( list ) | ID ;
list  :  list , elem | elem ;
```

we then get the following in `y.output`

`Grammar`	*Here all the rules are given.*

```
rule 1    elem -> ID '(' list ')'
rule 2    elem -> ID
rule 3    list -> list ',' elem
rule 4    list -> elem

Terminals, with rules where they appear
```

`$ (-1)`	*$ represents the end token.*
`'(' (40) 1`	*Each token is given a token id.*
`')' (41) 1`	
`',' (44) 3`	
`error (256)`	
`ID (258) 1 2`	

```
Nonterminals, with rules where they appear


elem (7)
    on left: 1 2, on right: 3 4
list (8)
    on left: 3 4, on right: 1 3
```

`on left` means on left side of `->` in the `Grammar` section

```
state 0
    ID          shift, and go to state 1
    elem        go to state 8

state 1
    elem  ->  ID . '(' list ')'   (rule 1)
    elem  ->  ID .   (rule 2)
    '('         shift, and go to state 2
    $default    reduce using rule 2 (elem)

state 2
    elem  ->  ID '(' . list ')'   (rule 1)
    ID          shift, and go to state 1
    elem        go to state 3
    list        go to state 4

state 3
    list  ->  elem .   (rule 4)
    $default    reduce using rule 4 (list)

state 4
    elem  ->  ID '(' list . ')'   (rule 1)
    list  ->  list . ',' elem   (rule 3)
    ')'         shift, and go to state 5
    ','         shift, and go to state 6

state 5
    elem  ->  ID '(' list ')' .   (rule 1)
    $default    reduce using rule 1 (elem)

state 6
```

```
    list  ->  list ',' . elem   (rule 3)
    ID          shift, and go to state 1
    elem        go to state 7

state 7
    list  ->  list ',' elem .   (rule 3)
    $default    reduce using rule 3 (list)

state 8
    $           go to state 9

state 9
    $           go to state 10

state 10
    $default    accept
```

A.3 Comments

A.3.1 Semantic actions

It is possible to write a semantic action inside a grammar rule. This will cause the semantic action to be performed before the entire rule is being reduced. Bison does this by replacing the semantic action with a production rule that reduces to the empty string. Example, the rule

```
start : ONE { $$ = 42; }
        two { $$ = 42; }
        three { $$ = $2 + $4; }
      ;
```

will be replaced by Bison with the two rules

```
start : ONE @1 two @2 three { $$ = $2 + $4; } ;
@1    : { $$ = 42; } ;
@2    : { $$ = 42; } ;
```

This means that in the first rule (`start`), \$2 will have the value which is returned by the new rule `@1` (in this case 42) plus the value returned by the new rule `@2` (also 42 in this case). This trick is very useful when you want to generate labels and be able to reference them later in the code generation. Especially note the following:

- `@1` and `@2` are empty rules. They add nothing to the grammar. Their only purpose is to execute a semantic action when being reduced.
- Bison generates a new rule for each semantic action, even if they are equal. If you have several instances of equivalent semantic rules you may always construct your own empty rule to execute that action.

Types of semantic actions

Whenever we reference an item in a rule it have to have a defined *type*. We know how to specify the types of rules (using the `%type` command) and for tokens (by using the `%token` command). How do we specify the type of a semantic action?

The answer is that we can force any `$`-expression (`$$` or `$`*n*) to a type by inserting a <*type*> after the `$` (`$`<*type*>`$` or `$`<*type*>*n*, respectively). This means that the above example acually should be written:

```
start : ONE
          { $<integer>$ = 42; }
        two
          { $<integer>$ = 42; }
        three
          { $$ = $<integer>2 + $<integer>4; }
      ;
```

assuming that we have previously declared `integer` as the type for integers and declared `start` to be of type `integer`.

A.3.2 Common errors

Some of the more common errors are described here.

Forgetting to assign a value to $$

It is possible to forget to assign a value to $$ in very subtle ways. Here is *one* example. In a Bison specification the form:

```
name  :  ...
      |  ...
      ;
```

is only a shorthand for the the equivalent form:

name : . . . ;
name : . . . ;

This means that the first and the second instance of the rule are *completely* separated, there are *no* connection between them at all. Most importantly, if you assign something to a variable in one of the rules, it will *not* keep that value in the other rule. For example, the following rule will not work:

```
list    : TOKEN         { $$ = 1; }
        | list TOKEN    { $$++; }
        ;
```

Why? Since this form of rule is only a shorthand, we rewrite it to its full form.

```
list    : TOKEN         { $$ = 1; } ;
list    : list TOKEN    { $$++; }   ;
```

We can now see that the $$ that is referred is completely undefined in the second instance of the rule `list`. Therefore, what the second instance really does it to increase a random value.

Appendix B

The Symbol Table

When parsing a program you have to save information about the identifiers in the program, this is the purpose of the *symbol table*. The symbol table is a database containing the symbols and information about them. The symbol table is accessed through a number of functions and the functions are used during parsing by writing calls inside the semantic actions.

The types supplied can be seen in Figure B.1. To get access to the types, the user have to include the file `kompteo.h` , which also contains prototypes for the functions.

In addition to the symbol table, the library contains some functions that are useful when writing a compiler. These functions can be seen in Figure B.6.

B.1 Compiling with the symbol table library

The location of the library containing the symbol table and the the functions in Figure B.6 is `/stud/docs/kurs/kompteo/lib` and the name of the library is `libkompteo.a`. To instruct `gcc` that you want to use this library you have to supply the switch `-lkompteo` (`-l` means "use this library" and `kompteo` is the name of the library with the `lib` and `.a` removed).

To tell the compiler where it can find the library you use the `-L` switch followed by the name of the directory. The full switch then becomes

```
-L/stud/docs/kurs/kompteo/lib
```

In addition you have to tell the compiler where to find the non-standard include files you are using (namely `kompteo.h`), using the switch

```
-I/stud/docs/kurs/kompteo/include
```

Example: Assume that you have written a program `program.c`. The command to compile `program.c` is then:

```
gcc -I/stud/docs/kurs/kompteo/include\
   -L/stud/docs/kurs/kompteo/lib\
   program.c -lkompteo
```

This in *addition* to any switches that you have supplied when you have run Bison and/or Flex. If you are using Flex and/or Bison, the `-lkompteo` should be *before* `-lfl` and `-ly`. As an example, assume that you are writing a compiler and you want to link it all together. The full command is then:

```
gcc -I/stud/docs/kurs/kompteo/include\
    -L/stud/docs/kurs/kompteo/lib \
    lex.yy.c y.tab.c other.c -lkompteo -ly -lfl
```

This is quite a long command, which is why we have written a small script `compile` which you can copy. The command above and

```
./compile lex.yy.c y.tab.c other.c
```

does exactly the same. You can copy

```
/stud/docs/kurs/kompteo/bin/compile
```

to your directory (it is very small) and use it. Since there are two versions of the `libkompteo.a` library, depending on if you are running Solaris 5.4 or SunOS 4.1.3, the script will check what OS you are using and link in the appropriate library.

Following is a small example that inserts a function and a parameter (with parameter number and number of parameters respectively) in the symbol table and then print the contents of the symbol table on standard output (the screen).

```
#include <stdio.h>
#include <kompteo.h>

main() {
  insert_symbol("foobar", FUNCTION);
  set_no_params("foobar", 3);
  insert_symbol("gazonk", PARAMETER);
  set_paramno("gazonk", 1);

  print_symbols();
}
```

The resulting output when executing this code is:

```
gazonk : parameter #1
foobar : function with 3 parameters
```

Using markers

Markers are used in the symbol table to handle locally defined variables. When starting to parse the body of a function you will have to enter the parameters to the function as symbols in the symbol table. When you are through parsing the function those symbols should be removed again. For this you use *markers*. Whenever you start to parse a function you enter a marker into the symbol table using the function `set_marker`. When you are through parsing the function you remove all parameters to the function by calling `remove_marker`.

Example: To illustrate how the symbol table can be used when writing your compiler have a look at the following sequence of calls:

Function Call	**Symbol Table Contents**
`insert_symbol("f1", FUNCTION)`	f_1
`set_marker()`	f_1 †
`insert_symbol("a", PARAMETER)`	f_1 † a
`insert_symbol("b", PARAMETER)`	f_1 † $a\,b$
`remove_marker()`	f_1
`insert_symbol("f2", FUNCTION)`	$f_1\,f_2$
`set_marker()`	$f_1\,f_2$ †
⋮	⋮

Here † represents a marker and new symbols are added last in the symbol table.

```
typedef enum { UNDEFINED, FUNCTION, PARAMETER } Type;
typedef enum { FALSE, TRUE } Bool;
```

Figure B.1: Types of the symbol table

set_marker()

Description:
Set a marker in the symbol table.

remove_marker()

Description:
Remove the last set marker and all symbols that were inserted after it.

Figure B.2: Operations to handle markers

print_symbols()

Description:
Print all symbols in the symbol table with associated information. If the symbol is a function, the number of parameters will be printed; if the symbol is a parameter, the parameter number will be printed.

Figure B.3: Operations to print information about the symbol table

`Type` **get_type**(`char* name`)

Description:
Get the type of a symbol. If the symbol is undefined the function will return `UNDEFINED`.

`int` **get_paramno**(`char* name`)

Description:
Get the parameter number of a symbol, only works for parameter symbols. If the parameter is undefined, or the parameter number has not been set, return -1.

`int` **get_no_params**(`char* name`)

Description:
Get the number of parameters for the symbol, this only works for function symbols. If the function is undefined, or the number of parameters has not been set, return -1.

`char*` **get_last**(`Type type`)

Description:
Get the name of the last inserted symbol of type 'Type'. If there are no symbol of the type in the symbol table this operation will return NULL.

Figure B.4: Operations to retrieve information

`Bool` **insert_symbol**(`char* name, Type type`)

Description:

Add a symbol to the symbol table. If the symbol already is in the symbol table, the previous symbol will be "shadowed", but the insertions will not fail due to this.
If the symbol was successfully inserted, return `TRUE`, otherwise return `FALSE`.

`Bool` **set_paramno**(`char* name, int no`)

Description:

Set the parameter number of a symbol, only works for parameter symbols. If the parameter number for the symbol has been set before, the old value is overwritten with the new value.
If the referenced symbol is not a parameter, this operation fails.
If the parameter number was successfully set, return `TRUE`, otherwise return `FALSE`.
If an attempt to reset a parameter number is made, execution will abort with an error message.

`Bool` **set_no_params**(`char* name, int no`)

Description:

Set the number of parameters for the symbol, this only works for function symbols. If the number of parameters for the symbol has been set before, the old value is overwritten with the new value.
If the referenced symbol is not a function, this operation fails.
If the number of parameters was successfully set, return `TRUE`, otherwise return `FALSE`.
If an attempt to reset the number of parameter to a function is made, execution will abort with an error message.

Figure B.5: Operations to store information `int` **ceprintf**(`char* format, ...`)

Description:

Function to print a formatted error message. It works the same way as `printf(3)`, with the exception that `ceprintf` will print a message, preceded with a line number that it taken from the variable `Line_number` (see below).

int **Line_number**

Description:

This variable is used to remember the current line number. Initially it is set to 0, but otherwise it has to be changed manually whenever the line number changes.

`char*` **newlabel**(`char* pfx`)

Description:

Function to create new labels. Each time it is called, it generates a new label which has the prefix `pfx`.

Figure B.6: Library of Useful Functions

Appendix C

The Language AFL

AFL is a language with which arithmetic calculations can be performed. An AFL-program is made up of two parts. The first part contains function declarations and the second part defines the calculations which are performed by the program. The language is a so called functional language and contains no variables and thus no assignment statements. The following is an example of an AFL program:

```
! factorial calculates the factorial of a number recursively
factorial(n) := if n = 0 then 1 else n*factorial(n-1);

! cube calculates the cube of a number
cube(n) := n*n*n;

! pi defines the constant that is the proportion between
! the diameter and the circumference of a perfect circle.
pi := 3; ! Almost anyway

! Here comes the actual program, i.e. the definition of the
! calculations that will be performed
? cube(factorial(4)/pi)
```

C.1 Lexical definition of AFL

Table C.1 describes the different classes of lexemes (strings) that are included in the language AFL. The table also includes the name of the token and the symbolic name of the token you will use in the program you construct. A string is denoted as "...", where the quotes are not included in the string.

Token	**Symbolic name**	**Lexeme**
if	IF	”`if`”
then	THEN	”`then`”
else	ELSE	”`else`”
and	AND	”`and`”
or	OR	”`or`”
not	NOT	”`not`”
declaration	DECLARATION	”:=”
number	NUMBER	All strings of digits
id	ID	All strings of letters and digits, which contain at least one letter
(	LEFTPAREN	”(”
)	RIGHTPAREN	”)”
,	COMMA	”,”
;	SEMICOLON	”;”
$*$	STAR	”$*$”
/	SLASH	”/”
+	PLUS	”+”
$-$	MINUS	”$-$”
$>$	GREATER	”$>$”
$<$	LESS	”$<$”
=	EQUAL	”=”
?	COMPUTE	”?”

Table C.1: Tokens of AFL

C.1.1 Comments

Comments in AFL begin with an exclamation mark and include the rest of the line. Comments may appear anywhere on a line.

C.2 Syntactic definition of AFL

C.2.1 Operators

The language AFL includes three types of operators, arithmetic, relational and logical. The operators are given in table C.2 below in order of precedence, the first operator having the highest precedence. Two operators on the same line have the same precedence.

AFL also includes a conditional expression (also called `if-then-else` expression). A conditional expression must contain both a `then`-part and an

	Operator	**Associativity**
Arithmetic:	$*, /$	Left
	$+, -$	Left
Relational:	$>, <, =$	Left
Logical:	**not**	Right
	and	Left
	or	Left

Table C.2: Operators of AFL

`else`-part. In contrast to an `if`-*statement* in Pascal or C, a conditional expression in AFL has a value (as an example consider the function `factorial` in the example above), and can be a component in an arithmetic expression. The conditional expression has lower precedence than the other arithmetic operators.

C.2.2 Grammar

AFL can be defined with the following context free grammar (where nonterminals begin with a large letter and all tokens are in **bold face**):

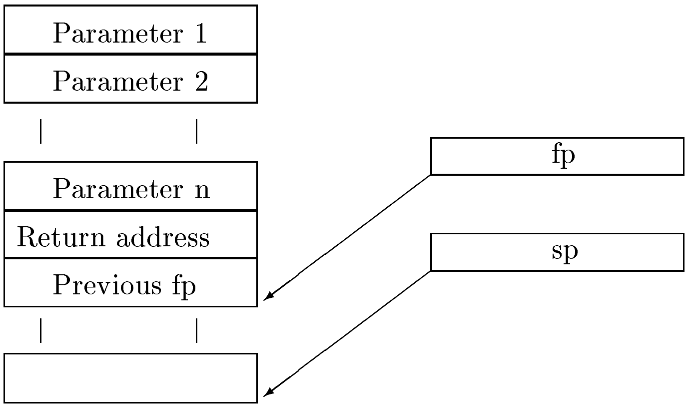
<table>
<tr><td>Program</td><td>$\rightarrow$</td><td>Declarations ? Expression</td></tr>
<tr><td>Declarations</td><td>$\rightarrow$</td><td>Declarations FunctionDef $\mid \epsilon$</td></tr>
<tr><td>FunctionDef</td><td>$\rightarrow$</td><td>id OptFormalList := Expression ;</td></tr>
<tr><td>OptFormalList</td><td>$\rightarrow$</td><td>$\epsilon \mid$ (FormalList)</td></tr>
<tr><td>FormalList</td><td>$\rightarrow$</td><td>id $\mid$ FormalList , id</td></tr>
<tr><td>Expression</td><td>$\rightarrow$</td><td>number $\mid$ id $\mid$ id (ActualList) $\mid$
Expression + Expression $\mid$
Expression - Expression $\mid$
Expression * Expression $\mid$
Expression / Expression $\mid$
if Condition then Expression else Expression $\mid$
(Expression)</td></tr>
<tr><td>ActualList</td><td>$\rightarrow$</td><td>Expression $\mid$ Expression , ActualList</td></tr>
<tr><td>Condition</td><td>$\rightarrow$</td><td>not Condition $\mid$
Condition and Condition $\mid$
Condition or Condition $\mid$
(Condition) $\mid$
Expression $=$ Expression $\mid$
Expression $>$ Expression $\mid$
Expression $<$ Expression</td></tr>
</table>

Table C.3: Grammar of AFL

Appendix D

The Abstract Machine AFM

AFM has three registers, *pc*, *sp* and *fp*. *pc* is the program counter register and contains the number of the next instruction. *sp* is a stack pointer which always points to the top of the stack. The stack grows toward lower addresses. *fp* is an activation record pointer register (frame-pointer), which points to the current activation record. The machine has no other memory than the stack. An AFM activation record is intended to have the following appearance:



Uppermost on the stack are the parameters of the function. Then comes the return address to the calling function and last is the previous value of the *fp*. Note that *fp* always points to the stack location two addresses after the last parameter, which means one can always access a parameter on the stack by an offset to *fp*, irrespective of the value of *sp*. The last parameter is always at address $fp + 2$, the next to last at address $fp + 3$, etc.

AFM instructions adhere to the following principle: First the operation is performed on the two uppermost elements on the stack, these elements are removed from the stack and finally the result of the operation is stored on the top of the stack.

Example: Assume that we want to compute the value of $3 + 5 * 8 - 32$. The following AFM code would compute the value of the expression to 11:

```
PUSH  3
PUSH  5
PUSH  8
MUL
ADD
PUSH  32
SUB
```

D.1 AFM instruction set

Instructions for moving data

`PUSH` and `PUSHIND` push data on the top of the stack. `PUSH` pushes a constant on the top of the stack, and `PUSHIND` copies an element from somewhere in the stack to the top of the stack. `POP` pops the element on top of the stack.

Arithmetic operations

`ADD`, `SUB`, `MUL` and `DIV` perform addition, subtraction etc. between the two top elements on the stack. These two elements are removed from the stack, and the result of the operation is pushed on to the top of the stack.

Logical operations

`AND` and `OR` perform logical and/or between the two top elements on the stack, similar to in the arithmetic operations. `NOT` performs a logical inversion of the top element on the stack.

Relational operations

`EQUAL`, `GREATER` and `LESS` compare the two top elements on the stack, and deletes these two elements from the stack. The result of the comparison (*true* or *false*) is stored on the top of the stack.

Jump instructions

`GOTO` causes an unconditional jump to an address. `GOTRUE` and `GOFALSE` causes a jump to an address if the top of the stack has the value *true* or *false* respectively. The top element is deleted from the stack.

Instructions for calling and returning

`FUNCALL` calls an address, and `FUNRET` deletes an activation field from the stack and returns from a function call. `FUNRET` takes as an argument the number of parameters of the function. These parameters are deleted from the stack when the return is performed. In addition, the value which is on the top of the stack will still be on the top of the stack after the return is performed. This value is assumed to be the value returned by the function.

The instruction `LINK` creates a new activation field on the stack, by first saving the value of fp on the stack and then setting fp to point to that place on the stack (as in the figure).

Pseudo-instructions

`END` indicates the end of a program. `TRACE ON` turns on the printing of each instruction before it is performed. In addition the contents of all registers and the stack is printed after every instruction is performed. The program waits after each instruction is performed for a carriage return. `TRACE OFF` turns this off.

`NOOP` is the instruction that does nothing at all.

The syntax for instructions and the operations which are performed in the machine for each instruction is given in table D.1 (page 50).

AFM instruction	*operations performed in the machine for each instruction*
PUSH n	$sp = sp - 1$ $\text{stack}[sp] = n$ $pc = pc + 1$
PUSHIND i	$sp = sp - 1$ $\text{stack}[sp] = \text{stack}[fp + i]$ $pc = pc + 1$
POP	$sp = sp + 1$ $pc = pc + 1$
SUB	$\text{stack}[sp + 1] = \text{stack}[sp + 1] - \text{stack}[sp]$ $sp = sp + 1$ $pc = pc + 1$
ADD, MUL, DIV	*in a similar manner*
AND	$\text{stack}[sp + 1] = \text{stack}[sp + 1]$ and $\text{stack}[sp]$ $sp = sp + 1$ $pc = pc + 1$
OR	*in a similar manner*
NOT	$\text{stack}[sp] = \text{not stack}[sp]$ $pc = pc + 1$
GREATER	if $\text{stack}[sp + 1] > \text{stack}[sp]$ then $\text{stack}[sp + 1] =$ `true` else $\text{stack}[sp + 1] =$ `false` $sp = sp + 1$ $pc = pc + 1$
EQUAL, LESS	*in a similar manner*
GOTO L	$pc = L$
GOTRUE L	if $\text{stack}[sp] =$ `true` then $pc = L$ else $pc = pc + 1$ $sp = sp + 1$
GOFALSE L	*in a similar manner*
FUNCALL L	$sp = sp - 1$ $\text{stack}[sp] = pc + 1$ $pc = L$
FUNRET n	$pc = \text{stack}[fp + 1]$ $\text{stack}[fp + n + 1] = \text{stack}[sp]$ $sp = fp + n + 1$ $fp = \text{stack}[fp]$
LINK	$sp = sp - 1$ $\text{stack}[sp] = fp$ $fp = sp$ $pc = pc + 1$

A symbolic address, “label”, is written first on each line, followed directly by a colon ‘:’.

D.2 Simulator

An AFM-simulator with which you can run your AFM programs can be found as `/stud/docs/kurs/kompteo/bin/afm`.

This simulator inputs an AFM-program and checks if it is correct. Then the program is run and the result is output (the result of a program is the value which is on the top of the stack when the program finishes executing). The simulator controls that no erroneous operations are performed, such as adding "true" to 10. To perform this control an element on the stack is represented by a type and a value. There are four types in AFM:

constant	Integer constants used in arithmetic operations
stack index	Stack references (*sp*and *fp*are of this type)
program index	Program references (*pc*is of this type)
boolean	True or False, used in logical operations

These types are always printed out when the "trace" is set. You do not need to generate these types yourself, it is done automatically in AFM.

To run the simulator use the following command

```
./afm filename
```

If you give the command `./afm -t filename` the trace will automatically be set. The command `./afm -l filename` writes out the AFM-program before it is executed.