FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Compilers Labo

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 27 april 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

Prologue

1.1 Using Docker

Each lab has a `files` directory which you should change your working directory to. Run the following command to start an interactive Docker container.

```
$ docker run --rm -it -v "$(pwd)":/files tbesard/compilers:practx
```

After the container has launched, you can navigate to the correct directory with

`cd files`

You can edit these files on your local host machine, but it is recommended to edit in the container (see next section).

1.2 Installing the nano editor

Editing files on a Windows computer brings incompatibility with linux tools. It is recommended to install a text editor on the container.

```
$ apt-get update
$ apt-get install nano
```

Hoofdstuk 2

Lexing

2.1 Setup

The goal of this lab is to process source files in a C-like language, and generate a stream of tokens along with location information, or an error message if the syntax of the file is invalid. This will be realised with the Flex lexical analyzer generator.

The project can be executed using the docker image `tbesard/compilers:pract1` (see Using Docker 1.1 for more information). Run `make` and execute the resulting `main` executable on a source file:

```
make && ./main test/dummy.c
```

2.2 Basic lexing

The first part is to add some new definitions and rules in `lexer.l`. The definitions consist of a label and a regex. The purpose of these definitions is to be used in rules. The rules match a regular expression and tells the lexer what to return.

```
/*
  Definitions
*/
DIGIT  [0-9]
WHITESPACE    [ \t\n\r]
ID      [a-z_][a-z0-9_]*
COMMENT    "//"[^\n\r]*
FLOAT     [0-9]+\.[0-9]*
STRING  \"[^\n]+\"


/*
  Rules
*/
"="          return EQUAL;
"=="         return CEQ;
"!="         return CNE;
"<"          return CLT;
">"          return CGT;
```

```
"<="         return CLE;
">="         return CGE;
"+"          return PLUS;
"-"          return MINUS;
"*"          return MUL;
"/"          return DIV;
"^"          return EXP;
"%"          return MOD;
","          return ',';
";"          return ';';
"("          return '(';
")"          return ')';
"{"          return '{';
"}"          return '}';
"["          return '[';
"]"          return ']';
"return"   return RETURN;
"if"       return IF;
"else"     return ELSE;
"while"    return WHILE;
"for"      return FOR;
{COMMENT}        /*must be ignored*/
{FLOAT}          return FLOAT;
{DIGIT}+         return INTEGER;
{WHITESPACE}+
{ID}             return IDENTIFIER;
{STRING}         return STRING;
.                error("unknown symbol");
```

2.3 Location information

To add location information, the method `update_location()` in `lexer.l` must be updated. This method will be called everytime the lexer returns a token from the rules. The struct `Location` in `lexer.hpp` contains two integers for the line number and column number. The `Lexer` class contains two `Location` structs which respectively represent the beginning line and column and the ending line and column of the token. The attribute `yytext` contains a string representation of the current token.

```
void Lexer::update_location() {
  if(yytext[0] == '\n' || yytext[0] == '\r'){
    begin.column = 1;
    end.column = 1;
    begin.line++;
    end.line++;
  } else {
    begin.column = end.column;
    end.column += strlen(yytext);
  }
}
```

2.4 Error reporting

To add error messages, the method `update_location()` in `lexer.l` must first be updated so that it keeps string information of the current line.

```
char *buffer = (char*)malloc(YY_BUF_SIZE);
void Lexer::update_location() {
  if(yytext[0] == '\n' || yytext[0] == '\r'){
    begin.column = 1;
    end.column = 1;
    begin.line++;
    end.line++;
    free(buffer);
    buffer = (char*)malloc(YY_BUF_SIZE);
  } else {
    begin.column = end.column;
    end.column += strlen(yytext);
    strcat(buffer, yytext);
  }
}
```

Additionaly, the method `error()` in `lexer.l` must be implemented. This method gets called when the lexer encounters an error.

```
void Lexer::error(const std::string &message) {
  std::cerr << "Syntax error: " << message
            << " at line " << begin.line
            << " at column " << begin.column
            << "\n";
  std::cerr << buffer << "\n";
  printf("%*c^\n", (int)begin.column - 1, ' ');
}
```

2.5 Context-sensitive rules

A context-sensitive rule are useful for patterns which cannot be matched with simple regular expressions. Here we add the neccesary definitions and rules to parse block comments. We add a definition `BLOCK_COMMENT`, which is preceded by `%x`. This symbol means that `BLOCK_COMMENT` is an exclusive state, which means that the lexer will only match rules which are tagged `BLOCK_COMMENT` once it enters the state. Next a new set of rules is implemented. The `INITIAL` state is a predefined state which marks the entry point for any other state. We specify that a block comment starts with /*, and if that pattern occurs, the state `BLOCK_COMMENT` is activated. Only patterns in the `BLOCK_COMMENT` state block will be matched. Now four possible rules can be matched:

1. The end of a comment is specified with */. If this occurs the comment has succesfully ended and we go back to the initial state.
2. If it's not the end of the coment, there could be an infinite number of characters we have to eat ([^*\n]+). We do not include the newline character because we want it seperately to increase the line number.
3. When we encounter a newline, we want to increase the line number of the lexer.

4. When the end of a file is reached, the block comment was not terminated, resulting in an error message.

```
/*
  Definitions
*/
%x BLOCK_COMMENT

/*
  Rules
*/
<INITIAL>{
  "/*" BEGIN(BLOCK_COMMENT);
}
<BLOCK_COMMENT>{
  "*/"   BEGIN(INITIAL);
  [^*\n]+
  \n     yylineno++;
  <<EOF>> {
    error("unterminated block comment");
    BEGIN(INITIAL)
  }
}
```

Hoofdstuk 3

Parsing

3.1 Setup

The goal of this lab is to complete the implementation of a parser for the same C-like language we have worked with in the first lab. This will be realised with the Bison parser-generator tool in order to generate the BNF grammar that implements the rule of this language.

After mounting the image, configure the project using

```
cmake .
```

This only need to be done once. To actaully compile the files, use

```
make
```

After compiling, a binary named `cheetah` will be created. This binary will visualize the AST of a source file in GraphViz DOT format. This DOT format can be rendered to a PNG.

```
./cheetah ../test/dummy.c > dummy.dot && dot -Tpng dummy.dot > dummy.png
```

3.2 Function call

We will first implement the function call expression. The rule needs to produce an expression, and as such needs to be part of the `expr` production. A function call is represented by a `AST::CallExpr` object and the argument list is of type `AST::ExprList`. In `parser.y` we first add two new representations to the `%union` clause.

```
%union {
  ...
  ...
  AST::CallExpr *call_expr_t;
  AST::ExprList *expr_list_t;
}
```

These symbols also need to be classified as a nonterminal symbol.

```
%type <call_expr_t> func_call
%type <expr_list_t> expr_list
```

3.3 Literals

3.4 Operators

3.5 Control flow

Hoofdstuk 4

Code Generation

4.1 Setup

Run a docker container and configure the project with CMake.

```
$ docker run --rm -it -v "$(pwd)":/files tbesard/compilers:pract3
$ cd /files
$ cmake .
```

Run `make` to compile the whole project after each change. Use `cheetah` to generate the assembly code.

```
$ ./cheetah test/dummy.c
.globl main
main:
  pushq $1
  popq %rax
  ...
```

Een executable aanmaken kan met `make dummy`, of als je alle testen wilt compileren kan je `make test` gebruiken.

4.2 Debugging

Met `gdb` kan een executable geïnspecteerd worden.

```
$ gdb ./test/dummy
(gdb) run
```

4.3 Compiler infrastructure

The `codegen.hpp` header defines three important datastructures:

- `Program`: This represents the program that is being emitted, and is accesible as the argument to each `emit` function. It contains a list of `Blocks`.

- `Block`: A block is identified by a label and contains a list of `Instructions`.
- `Instruction`: An instruction has three fields:
  - `name`: the textual representation of the instruction name.
  - `arguments`: a list of arguments.
  - `comment`: an optional string that will be emitted as part of the generated code.

4.4 Emitting code

We will implement the compiler as a stack machine. This means that it should push and pop values onto the stack and only use registers when absolutely neccesary. An explanation of the most useful registers:

- `%rax`: Temporary register, mainly used as the return register.
- `%rbx`: Callee-saved register which can optionally be used as a base pointer.
- `%rbp`: Callee-saved register which can optionally be used as a frame pointer.
- `%rdi`: Used to pass the first argument to functions.
- `%rsi`: Used to pass the second argument to functions.
- `%rdx`: Used to pass the third argument to functions. Can also be used as the second return register.
- `%rcx`: Used to pass the fourth argument to functions.
- `%r8`: Used to pass the fifth argument to functions.
- `%r9`: Used to pass the sixth argument to functions.
- `%r12-r15`: Callee-saved registers.

A short summary of the special registers:

- **Stack Pointer**: The stack pointer points to the top of the stack. All locations beyond the stack pointer are considered to be garbage, and all locations before the stack pointer are considered to be allocated.
- **Frame Pointer**: The area on the stack devoted to local variables, parameters, return address and other temporaries for a function is called the function's `stack frame`. The frame pointer points at the beginning of a stack frame such that the stack pointer can be restored to the frame pointer. Equivalently, the frame pointer contains the value of the stack pointer just before a function is called.
- **Base Pointer**: The base pointer is derived from the stack pointer and is used to travel trough the stack.

Each AST object now has an `emit` function, which purpose is to generate assembly code for that AST object.

4.4.1 Function calls

In `emit.cpp`, complete the implementation of `CallExpr::emit(Program &prog) const` and implement the following features:

- **emit and store arguments**

```
for (size_t i = 0; i < argc; i++){
  args[i]->emit(prog)
}
```

  Remember that this is a method of the class `CallExpr`, so we can use the attributes `args` and `name` of this class. The attribute `args` is of type `ExprList`, which can contain pointers to various expression types such as `FloatLiteral` or `Assignment` (see `expr.hpp`). We will simply call the emit method for each `Expr` in this list.

- **generate a call**

```
prog << Instruction{"call", {name->string}};
```

  Here we need to emit a `call` instruction. An instruction has a `name`, a list of `arguments` and optionally a `comment`. The name of the instruction is obviously `call`. In this case the list of arguments only contain one element: the name of the function. We opted to not include a comment here since a `call` instruction is fairly obvious.

- **return a value**

```
if(std::get<0>(decl) == T_void){
  prog << Instruction{"pushq", {"0xABCDEF"}, "void return value"}
} else if(std::get<0>(decl) == T_int){

}
```

  Our language only has two possible return types: `void` and `int`.

4.4.2 Function Declarations

In `emit.cpp`, complete the implementation of `FuncDecl::emit(Program &prog) const` and implement the following features:

- **The function prologue**:
  - save callee-saved registers
  - set the base pointer
  - align the stack pointer by 16 bytes
- **The function epilogue**:
  - restore the stack pointer
  - restore callee-saved registers