Labo 1: lexing

Doel: starten van broncode en alle tokens hieruit halen en waar deze voorkomen. Error messages genereren qua syntaxfouten

Gebruik van **flex**: lexicale analyse generator

make\_tarball.sh => gebruiken om project in te dienen

./make\_tarball.sh 05 "Bert De Saffel & Xandro Vermeulen"

## Docker

*docker run --rm -it -v /c/Users/xandr/lexer\_files:/pract1 tbesard/compilers:pract1*

bestaande map $(pwd):/pract1 mappen op tbesard/compilers:pract1 en komt daar in pract1 terecht

tbesard/compilers:pract1 is de image (stuk na ‘:’ is een tag voor welke versie uit zijn dockerhub je neemt)

## algemeen

uitvoeren: make && ./main test/dummy.c die c klasse meegeven als argument aan de /main

ctrl+d voor uit een docker sessie te gaan

## Lexer.l

Notes:

yytext bevat de string waarde van de laatst ingelezen TOKEN

{newline} moet ook in brackets zitten, niet zonder brackets doen dus aje geen ‘+’ ofzo nodig hebt

Begin.column = end.column het gaat van waar vorige lijn gestopt is

## Basic lexing

**lexer.l**:

Definitie toevoegen:

DIGIT [0-9]

Regel toevoegen:

{DIGIT}+ {return INTEGER}

* INTEGER wordt geinclude door lexer.hpp
* Hetgeen wat in DIGIT staat 1 of meer keer (+)

Voor whitespace [ \t\s] moet er ook een definitie/regel zijn of het wordt niet geabsorbeerd en dan zal hij klagen over een unknown symbol found.

## Location Information

* Wordt uitgevoerd voor elke uitvoering van een lijntje bij de ‘regels’
* In lexer.hpp is er een location struct waarin line (huidige lijn) en column (huidig karakter op die lijn) wordt bijgehouden voor zowel begin als einde van een token.
* Aanroepen in de code gewoon via begin.line / begin.column / end.line / end.column
* De huidige gevonden token vind je via het beschikbaar attribuut yytext

## Error reporting

* YY\_BUF\_SIZE kan je gebruiken als bufsize
* Niet vergeten <memory.h> te includen voor malloc te kunnen doen
* Niet vergeten bij malloc het naar (\*char) te casten, same met de buffer leeg te maken
* Want buffer bijhouden voor de huidige regel code bij te houden
* Printf(“%\*c^”,(int)n,’ ‘) zal n spatiekarakters printen en dan een ^

## Context-sensitive rules

* In *Definitions* moet de definitie van een **toestand** gedefinieerd worden, zodat die kan gebruikt worden in *Rules*. Algemene vorm:  
  %x <NAAM>. Wij willen een toestand hebben voor als we in een block comment zitten:  
    
  %x betekent dat indien de huidige toestand BLOCK\_COMMENT is, enkel regels die bij die toestand horen geldig zijn. Je kan ook %s gebruiken, zodat alle regels nog steeds geldig zijn.
* In *Rules* kan gebruik gemaakt worden van de **INITIAL** toestand, die door flex zelf al gedefinieerd is:  
    
  De **BLOCK\_COMMENT** toestand is dan  
  

Labo 2: parsing

Gebruiken van Bison om een parser te genereren.

Documentatie Bison: <https://users.elis.ugent.be/~tbesard/compilers/bison/>

We gaan vooral **parser.y** aanpassen

*Eigen script gemaakt:* src2tree.sh <FILE>

Geef een bestand mee ZONDER extensie, en ZONDER map te specificeren

Bv : src2tree.sh fibonacci

## Setup

**Op host toestel**

cd /path/to/**parser\_files**

*docker run --rm -it -v “$(pwd)”:/pract tbesard/compilers:pract2***Op virtuele machine**

Naar folder navigeren  
 *cd pract*

Code configureren (slechts één maal uitvoeren):   
 *cmake .*

Code compileren (altijd uitvoeren als ge wilt testen)  
 *make*

Abstract Syntax Tree visualiseren (voor dummy.c):  
 *./cheetah test/dummy.c > dummy.dot && dot -Tpng dummy.dot > dummy.png*Dit zal een .png bestand genereren die ge dan kunt openen op uw host toestel

## Function Call

AST::CallExpr \*call\_expr\_t;

AST::ExprList \*expr\_list\_t;

toevoegen in **parser.y** in de **%union { … }**

Dan ook

%type <call\_expr\_t> func\_call

%type <expr\_list\_t> expr\_list

toevoegen bij de lijst van %type

Deze func\_call kan nu gebruikt wordenom een grammaticaregel op te stellen. Voorbeeld van bestaande productieregel:

stmt:

';' { $$ = nullptr; }

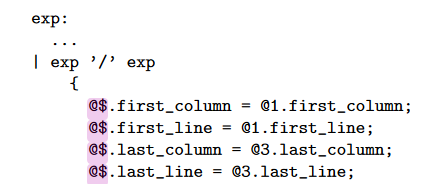
| expr ';' { $$ = $**1**; }

| decl { $$ = new **AST::DeclStmt**($**1**);

$$->location = @$; }

;

Dit zegt dat een **stmt** drie dingen kan zijn :

1. Een **;** zorgt ervoor dat een stmt geen semantische waarde heft
2. Een **expr** gevolgd door een **;**  
   De semantische waarde van **stmt** wordt die van het **1ste** type, **expr**, dus **$1**
3. Een **decl**  
   De semantische waarde van een **stmt** wordt nu een referentie naar een **DeclStmt**  
   De pseudovariabele **@$** bevat de locatie (lijn en kolom) van de linkergroep. Default is de start van deze waarde de locatie van het eerste symbol en einde de locatie van het laatste symbool  
     
   Bovenstaande code is default en moet nooit zelf gedeclareerd worden

**De functiecall productie:**

func\_call:

ident '(' ')' { *// functie zonder parameters: foo();*

$$ = new **AST::CallExpr**($**1**);

$$->location = @$;

} |

ident '(' expr\_list ')' { *// functie met oneindig veel parameters: foo(x, bar(zoo()), y, z, ...)*

$$ = new **AST::CallExpr**($**1**, \*$**3**); *// pointer naar $3*

$$->location = @$;

};

expr\_list:

%empty {

$$ = sema.**ParseExprList**();

} |

expr {

$$ = sema.**ParseExprList**($**1**);

} |

expr\_list ',' expr {

$$ = sema.**ParseExprList**($**3**, $**1**);

};

## Literals

Uitbreiden **semaexpr.cpp** (de methode signatuur wordt gegeven in **sema.hpp**, dus daar heb ik de parameters van overgenomen)

AST::IntLiteral \***Sema::ParseIntLiteral**(const Location &Loc, std::string IntToken) {

long long parsed = **strtol**(IntToken.**c\_str**(), nullptr, **0**);

*// if(parsed == 0L){*

*// throw semantic\_error(Loc->location, "invalid conversion");*

*// }*

IntLiteral \*IntLiteral = new **AST::IntLiteral**(parsed);

IntLiteral->location = Loc;

return IntLiteral;

}

AST::FloatLiteral \***Sema::ParseFloatLiteral**(const Location &Loc, std::string FloatToken) {

float parsed = **strtof**(FloatToken.**c\_str**(), nullptr);

*// if(parsed == 0.0) {*

*// throw semantic\_error(Loc->location, "invalid conversion");*

*// }*

FloatLiteral \*FloatLiteral = new **AST::FloatLiteral**(parsed);

FloatLiteral->location = Loc;

return FloatLiteral;

}

Uitbreiden **expr** productie:

STRING {

$$ = new **AST::StringLiteral**(**yytext**(lexer));

$$->location = @$;

} |

FLOAT {

$$ = sema.**ParseFloatLiteral**(@$, **yytext**(lexer));

} |

INTEGER {

$$ = sema.**ParseIntLiteral**(@$, **yytext**(lexer));

}

### Operators

Uitbreiden **expr** productie

expr CEQ expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CEQ, $**3**);

$$->location = @$;

} |

expr CNE expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CNE, $**3**);

$$->location = @$;

} |

expr CLT expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CLT, $**3**);

$$->location = @$;

} |

expr CLE expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CLE, $**3**);

$$->location = @$;

} |

expr CGT expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CGT, $**3**);

$$->location = @$;

} |

expr CGE expr {

$$ = new **AST::BinaryOp**($**1**, Operator::CGE, $**3**);

$$->location = @$;

} |

expr PLUS expr {

$$ = new **AST::BinaryOp**($**1**, Operator::PLUS, $**3**);

$$->location = @$;

} |

expr MINUS expr {

$$ = new **AST::BinaryOp**($**1**, Operator::MINUS, $**3**);

$$->location = @$;

} |

expr MUL expr {

$$ = new **AST::BinaryOp**($**1**, Operator::MUL, $**3**);

$$->location = @$;

} |

expr DIV expr {

$$ = new **AST::BinaryOp**($**1**, Operator::DIV, $**3**);

$$->location = @$;

} |

expr MOD expr {

$$ = new **AST::BinaryOp**($**1**, Operator::MOD, $**3**);

$$->location = @$;

} |

expr EXP expr {

$$ = new **AST::BinaryOp**($**1**, Operator::EXP, $**3**);

$$->location = @$;

} |

PLUS expr {

$$ = new **AST::UnaryOp**(Operator::PLUS, $**2**);

$$->location = @$;

} |

MINUS expr {

$$ = new **AST::UnaryOp**(Operator::MINUS, $**2**);

$$->location = @$;

}

Definiëren **associativiteit**

%nonassoc EQUAL

%nonassoc CEQ CNE CLT CLE CGT CGE

%left PLUS MINUS

%left MUL DIV MOD

*// unary operators??*

%right EXP

## Control flow

Labo 3: code generation

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.

Debugging

Met gdb kan een executable geïnspecteerd worden.

$ gdb ./test/dummy

(gdb) run

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.

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.

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.

Labo 4: IR optimalisatie