Overview of Computer Architecture

High Level Languages

Carl Henrik Ek - carlhenrik.ek@bristol.ac.uk

December 13, 2019

http://carlhenrik.com

High Level Languages

1. *Lexer/Tokeniser*
2. *Parser*
3. Translator
4. Optimiser
5. Code Generator

Lexer

Input sequence of characters

Output sequence of tokens with

- `type` (`KEYWORD`, `WORD`, `LPAREN`, ..)
- `value` ([`WORD` →"main"], [`LPAREN` →"("]
- debugging info (file, line, position)

Operation recognise tokens with state machines, can catch tokens that doesn't exists

Code

```
int a(int *b){return 0;}
int a) int int *b( {return -1;}
```

- Both of these are valid to the `lexer`
- But only the top is valid `C` code

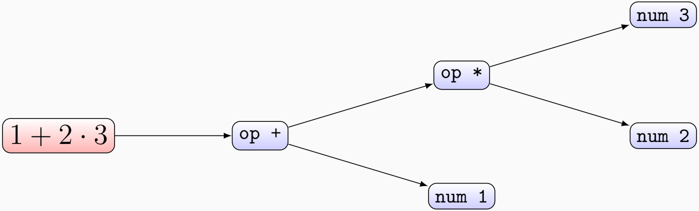
Parser

Input sequence of tokens

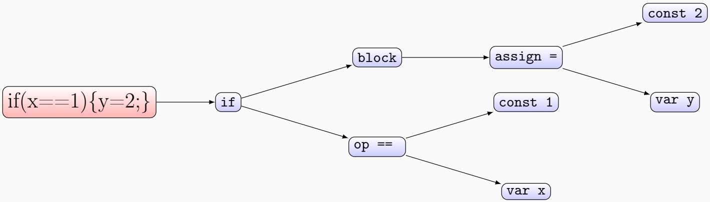
Output syntax tree

Operations very language dependent

Syntax Trees



Syntax Trees

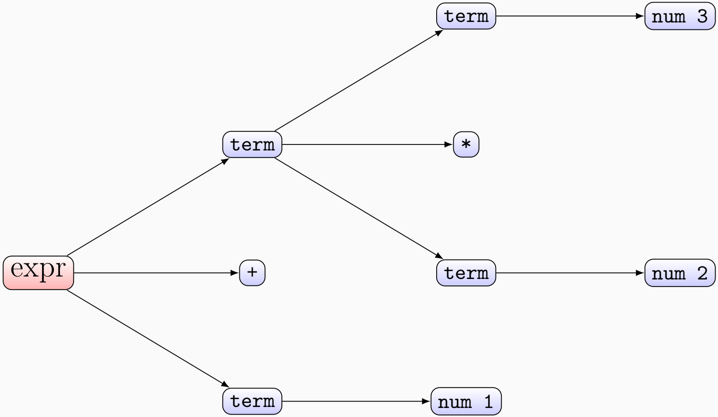


Grammars

$$1 + 2 \cdot 3$$

- How can we parse this sentence?
- How can we evaluate this?

Grammars



- `expr: term '+' term | term | num`
- `term: term '*' term | num | '(' expr ')'`

C Grammar

stmt expr';' | cond | block | ..

cond if'('expr')'stmt

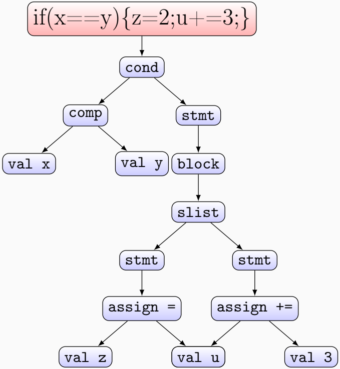
block '{'slist'}'

slist stmt | slist stmt

expr exrp assign expr | expr comp expr | val

assign '' | '+' | '-=' | ...

comp '=' | '!' | '>'



Why do languages look the way they do?

$$1 + 2 \cdot 3$$

$$1\ 2\ 3\ \cdot\ +$$

$$2\ 3\ \cdot\ 1\ +$$

- "Normal notation requires involved syntax tree"
- Reverse Polish Notation directly written as tree

Grammars

- We can check if the code matches the defined grammar of the language
- If we have the associated information of where in the file the token comes from we can potentially output error

Grammars

- We can check if the code matches the defined grammar of the language
- If we have the associated information of where in the file the token comes from we can potentially output error
- *Why are GCCs error messages often not at the right line?*

Continue

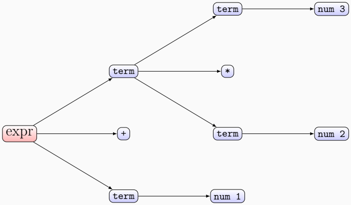
Translator

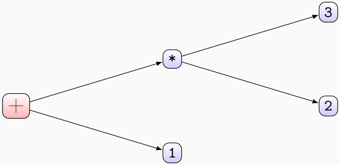
Input Syntax Tree

Output An independent representation (IR) (AST)

Operations symbol tables, tree transformations, semantic analysis

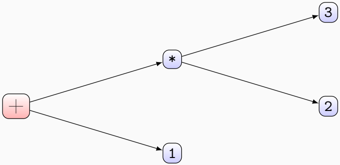
Tree Transformation





Semantic Analysis

- `eval(n) = n`
- `eval(+) = eval(a) + eval(b)`
- `eval(*) = eval(a) * eval(b)`



Evaluation

`eval(add(num 1)(mul(num 2)(num 3))) = eval(num 1) + eval(mul(num 2)(num 3)) = 1 + (eval(num 2) * eval(num 3)) = 1 + (2 * 3)`

Syntax and Semantics

Syntax structure of code

- "The circle square"

Semantics meaning of code

- "The circle is square"

Syntax

Code

```
int main(void)
{
  a = 3;
  int a
  int b = 1;

  return -1;
}
```

Symbol Tables

Code

```
int main(void)
{
  int a;
  a = 3;
  int b = 1;

  return -1;
}
```

Symbol Tables

Code

```
int x; /*a declaration - goes into the symbol table*/
x = 1; /*a definition - produces machine code*/
```

`C` requires you to declare names (functions, variables, etc) before you use them.

Code

```c
void f()
{
  char x = 2;
  if (x)
    {
      int x = 3;
      fprintf(stdout, "%d\n", x);
    }
}
int main(void)
{
  long x = 1;
  f();
}
```

Scopes

- Why would you want nested variable declarations?
- How to implement nested variable declarations?
- Why would you want to separate declarations and definitions?

Typing

In C `x++` means,

add 1 to `x` if `x` is an integer

add `sizeof(T)` **to** `x` if `x` is a `T*`

an error if `x` is a `struct`

`x+y` needs different instructions in the implementation dependent on type

Different types of typing

Static Typing variables are established ad compile time and cannot change.

Dynamic Typing the type can change during run-time

Dynamic Typing

Code

```
x = 2.0
print(type(x))
x = 'apa'
print(type(x))
```

Asymmetries

$$x = y$$

$$y = x$$

L and R values

- L and R values interact through assignment operators (=)
- L-values refers to memory locations that identifies an object
- R-values refers to values that are stored in memory. We cannot assign a value to a r-value

L - Values

- The name of a variable of any type

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-inderiction (*) that does not refer to a pointer

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-inderiction (*) that does not refer to a pointer
- An l-value expression in parentheses

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indericrtion (`*`) that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indericiton `(*)` that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object
- The result of indirection through a pointer (not function pointer)

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indericiton `(*)` that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object
- The result of indirection through a pointer (not function pointer)
- Member access in a struct `->`

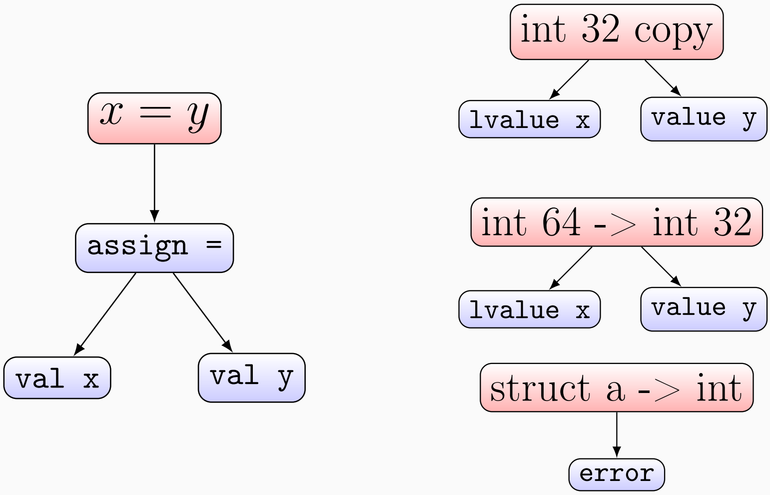
Examples

Code

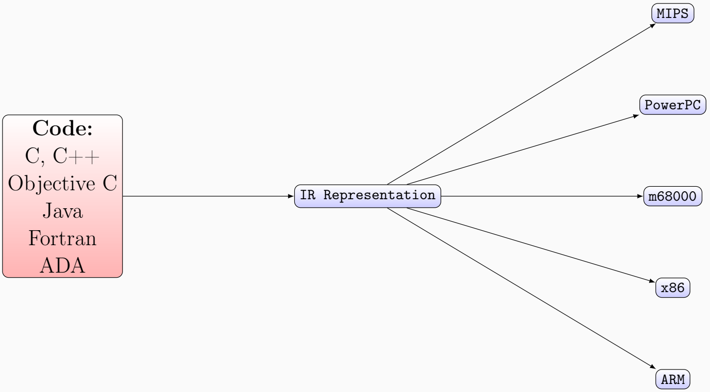
```
int a = 1, b;
int *p, *q;

b = a;
*p = 1;
q = p+5;
a + 1 = b;
p = &a;
&a = p;
```

Translator


$x = y$
assign =
val x
val y
int 32 copy
lvalue x
value y
int 64 -> int 32
lvalue x
value y
struct a -> int
error

Intermediate Representation



Translator

- Transform syntax tree
- Create Symbol Table
  - Deal with scopes
  - Deal with types
- Normally outputs an intermediate hardware independent representation
  - Abstract Syntax Tree (AST)

High Level Languages

1. Lexer/Tokeniser
2. Parser
3. Translator
4. Optimiser
5. Code Generator

Optimiser

Input Abstract Syntax Tree

Output Optimised AST

Operations elimenate dead code, eliminate repeated register assignments, etc

Example

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return 0;
}
```

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return 0;
}
```

Code

```
mov     r3, #3
str     r3, [fp, #-8]
mov     r3, #4
str     r3, [fp, #-12]
ldr     r2, [fp, #-12]
ldr     r3, [fp, #-8]
add     r3, r2, r3
str     r3, [fp, #-12]
mov     r3, #0
mov     r0, r3
```

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return 0;
}
```

Code

```asm
mov         r0, #0
b         lr
```

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return y;
}
```

Code

```
mov     r3, #3
str     r3, [fp, #-8]
mov     r3, #4
str     r3, [fp, #-12]
ldr     r2, [fp, #-12]
ldr     r3, [fp, #-8]
add     r3, r2, r3
str     r3, [fp, #-12]
ldr     r3, [fp, #-12]
mov     r0, r3
```

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return y;
}
```

Code

```
mov       r0, #3
mov       r1, #4
add       r0, r0, #0
add       r1, r1, r0
mov       r0, r0, r1
b       lr
```

Code

```c
int main(void)
{
  int x;
  int y;
  x = 3;
  y = 4;

  x += 0;
  y += x;

  return y;
}
```

Code

```
mov         r0, #4
add         r0, r0, #3
bl
```

Code

```c
int main(void)
{
  int y = 0;
  for(int i;i<100;i++)
    {
      y+=1;
    }
  return y;
}
```

Code

```
eor     r0, r0, r0
mov     r7, #100-1
loop:
add     r0, r0, #1
subs     r7, r7, #1
bne     loop
```

1 eor, 1 mov, 99 add, 99 subs, 99 bne

Code

```c
int main(void)
{
  int y = 0;
  for(int i;i<100;i++)
    {
      y+=1;
    }
  return y;
}
```

Code

```
eor        r0, r0, r0
add        r0, #1
add        r0, #1
..............
add        r0, #1
```

- 1 eor, 99 add
- *probably 3 times faster*
- gcc -funroll

Register

Code

```
mov     r3, #4
str     r3, [fp, #-8]
ldr     r3, [fp, #-8]
lsr     r2, r3, #31
```

- Memory access is a lot more expensive than registers access
    - reduce memory access and keep using registers

Execution

Moder CPUs execute in complex manners

- caches
- pipelines
- branch predictors
- vector instructions
- .....

A good optimiser should take all this into account

Optimiser

- The optimisation is very involved
  - GCC have over 150 passes through the code for the full proceedure
- Interesting to read up what the different flags are doing [1]

[1] https://www.linuxjournal.com/article/7269

High Level Languages

1. Lexer/Tokeniser
2. Parser
3. Translator
4. Optimiser
5. Code Generator

Code Generator

Input optimised AST

Output machine code/executable file (if linked)

Operations convert AST

- AST $\rightarrow$ assembly
- assembly $\rightarrow$ machine code

Code

```c
long x = 1; /*Global*/
void f(void)
{
  /*PUSH to make space on Stack*/
  char x = 2;
  if (x)
    {
      /*PUSH*/
      int x = 3;
      fprintf(stderr, "%d\n", x);
    }
  /*POP*/
}
/*POP*/
```

Local Variables

- Local variables live on the stack
  - PUSH when they enter scope
  - POP when they go out of scope
- *think recursive calls to a function*

Local Variables

Code

```
824c:   e52db004   push   {fp}
8250:   e28db000   add    fp, sp, #0
8254:   e24dd00c   sub    sp, sp, #12
.............................................
8280:   e24bd000   sub    sp, fp, #0
8284:   e49db004   pop    {fp}
8288:   e12fff1e   bx     lr
```

Cross Compiler

Code

```
> arm-none-eabi-as -o tst.o -g tst.s
> arm-none-eabi-ld -o tst tst.o
> qemu-arm -singlestep -g 1234 tst &
> arm-none-eabi-gdb
```

Summary

1. Lexer/Tokeniser
2. Parser
3. Translator
4. Optimiser
5. Code Generator

Summary

- Compilers are massive projects, look at the source code for `gcc`
- Each step is split into several different phases
- Understand the basics and the motivation
  - what is it that we need to
  - what are the rough steps
  - what is hard/challenging with each part

Summary

What have we hoped that you have

- Understand how a computer works
- Lose respect for the computer
- Learn how to directly interact with hardware
- Understand how abstractions work
  - Operating system: execution/resource abstraction
  - High Level Languages: hardware

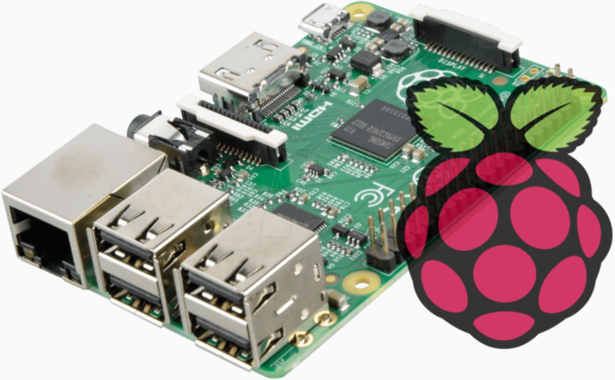
Lab Test

- 8 questions, 1h, multiple choice
- `arm-instructionset.pdf`
- code and execution of code
- you can use
  - the toolchain
  - not your previously written code
  - nor the internet

Exam

- everything but no ARM assembly
- code execution
- structure of CPU/Memory etc.
- Operating Systems basics
- High level languages/compilers

What to do next



eof