

Compiler Design - Cheatsheet

Compiler Structure

Simplified Compiler Structure

Source Code →

Lexical Analysis → Token Stream → Parsing → Abstract Syntax Tree

Front End

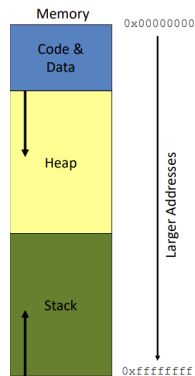
Intermediate Code Generation → Intermediate Code

Middle End

Code Generation

Back End

Stack layout



Suffixes

1. q = quadword (4 words)
2. l = long (2 words)
3. w = word (16-bit)
4. b = byte (8-bit)

movq-instruction

movq SRC, DST = DST ← SRC

Flags

e (equality)	ZF is set
ne (inequality)	(not ZF)
g (greater than)	(not ZF) and (SF = OF)
l (less than)	SF <> OF
ge (greater or equal)	(SF = OF)
e (less than or equal)	SF <> OF or Z

Addressing

-8 (%rsp)	rsp - 8
(%rax, %rcx)	rax + 8·rcx
8(%rax, %rcx)	rax + 8·rcx + 8

leaq vs. movq

In leaq we just compute the address, in movq we dereference the address.

Callee vs. Caller saved

Caller saved register (Rdi, Rsi, Rdx, Rcx, R09, R08, Rax, R10, R11) are saved by the caller before calling the function. Callee saved register (Rbx, R12, R13, R14, R15) are saved by the called function.

Parameter

1. 1...6: rdi, rsi, rdx, rcx, r8, r9
2. 7+: on the stack (in right-to-left order), nth arg.
 $((n - 7) + 2) \cdot 8 + rbp$

Why Intermediate Representations?

1. resulting code quality is poor (direct translation)
2. Richer source language features are hard to encode (Structured data types, Objects, ...)
3. hard to optimize
4. Control-flow is not structured

Basic Blocks

1. Starts with a label that names the entry point of the basic block
2. Ends with a control-flow instruction
3. Contains no other control-flow instructions
4. Contains no interior label used as a jump target

CFGs

1. Nodes are basic blocks
2. There is a directed edge from node A to node B if the control flow instruction at the end of block A might jump to the label of block B
3. No two blocks have the same label

getelementptr instruction

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. **GEP never dereferences the address it's calculating.**

```
struct RT {
    char A;
    int B[10][20];
    char C;
};
struct ST {
    int X;
    double Y;
    struct RT Z;
};
```

```
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
```

is translated in:

```
%arrayidx = getelementptr %struct.ST, ptr %s, i64 1,
i32 2, i32 1, i64 5, i64 13
ret ptr %arrayidx
```

Regular Expressions

Regular expressions precisely describe sets of strings. A regular expression R has one of the following forms:

1. ϵ : Epsilon stands for the empty string
2. 'a': An ordinary character stands for itself
3. $R_1 | R_2$: Alternatives, stands for choice of R_1 or R_2
4. $R_1 R_2$: Concatenation, stands for R_1 followed by R_2
5. R^* : Kleene star, stands for zero or more repetitions of R

Useful extensions:

1. "foo": Strings, equivalent to 'f' 'o' 'o'
2. R^+ : One or more repetitions of R, equivalent to RR^*
3. $R?$: Zero or one occurrences of R, equivalent to $(\epsilon | R)$
4. $[a' - 'z']$: One of a or b or c or ... z, equivalent to $(a|b|\dots|z)$
5. $[\sim 0' - '9']$: Any character except 0 through 9
6. R as x: Name the string matched by R as x

Chomsky Hierarchy

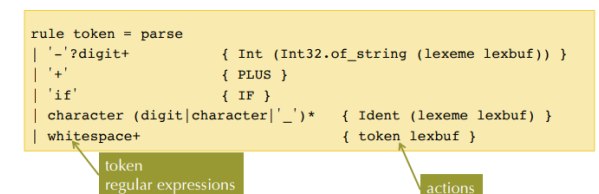
Regular \subset Context-Free \subset Context Sensitive \subset Recursively Enumerable

Matching Rule for Lexer

Most languages choose "longest match".

Lexer Generator

1. Reads a list of regular expressions: R_1, \dots, R_n , one per token
2. Each token has an attached "action" A_i (just a piece of code to run when the regular expression is matched)

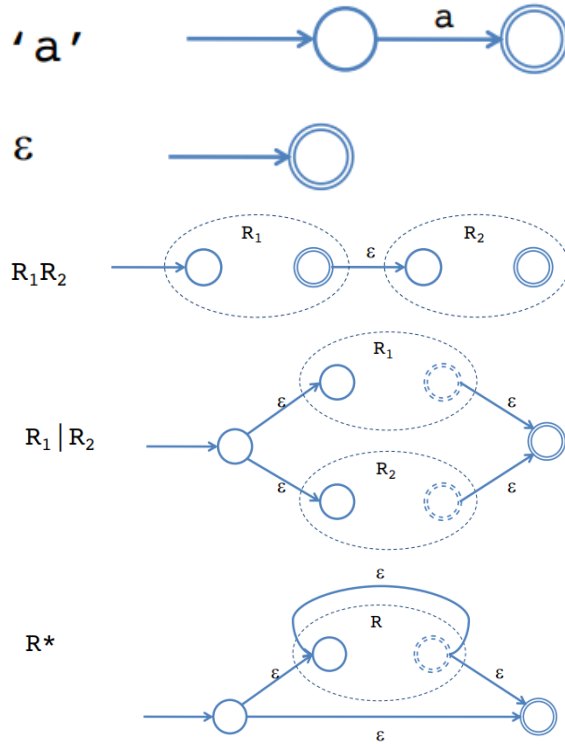


Generates scanning code that:

1. Decides whether the input is of the form $(R_1 | \dots | R_n)^*$
2. After matching a (longest) token, runs the associated action

Implementation Strategies

We can use a finite automaton since one must exist for a regular expression.



DFA vs. NFA

DFA:

1. Action of the automaton for each input is fully determined
2. Accepts if the input is consumed upon reaching an accepting state
3. Obvious table-based implementation

NFA:

1. Automaton potentially has a choice at every step
2. Accepts an input if there exists a way to reach an accepting state
3. Less obvious how to implement efficiently