07 Apr 2025 - Compilers -I - Week 02

syntax analyser = parser

annotated syntax tree = abstracted " "

transformations $\rightsquigarrow$ semantically equivalent

$\overset{\text{high-level}}{\text{input = stream}}$

Unofficially preprocessor

replace comments with blank

- Appendix, KR

  $\hookrightarrow$ 6 classes of tokens

→ Comments do not nest

→ Comments $\rightsquigarrow$ greedy algorithm

→ Preprocess

$$\texttt{/*} \quad \text{ok} \quad \underbrace{\texttt{/*}}_{\text{warning}} \quad \text{ok} \quad \texttt{*/} \quad \underbrace{\texttt{*/}}_{\text{error}}$$

```
# if 0
        ~~~~~~~~
        ~~~~~~~~
# endif
```

commented out

→ we will not build preproc.

→ Why modular?

↳ good for software design.

→ $$\left.\begin{array}{c}\text{LA}\\ \text{preprocessor}\end{array}\right\} \rightarrow$$ not a clear difference

→ preprocessor might do more work sometimes.

LA $\rightsquigarrow$ finite automata

→ <u>keywords</u>

→ first step: propose a language.

<u>identifiers</u> $l(l + d + \_)^*$

old C $\rightsquigarrow$ 31 char

→ array

new C $\rightsquigarrow$ stdstring

Will regex for identifier also match keywords?

Yes

Mini - languages

→ High - performance computing.

→ Token $\longrightarrow$ given by language designers

→ literal strings " . . . "

→ Pattern

→ Lexeme

→ Brackets LISP

( )

[ ]

< > ⟶ templates , (#include < > ⟶ preprocessor)

{ } ⟶ blocks

→ comma , colons , etc.

$$\begin{tikzcd}
\text{colons} \arrow[d] \\
\text{: ternary operator}
\end{tikzcd}$$

# include → keyword technically but eaten up by preproc.

IDE , code - completion

↳ make the ML model take and o/p tokens

↓

has a small abstract syntax tree to suggest code

09 Apr 2025

→ Missed extra class.

Little Languages

Domain-specific languages.

Regular Expressions

$$r = c^* (a \underset{\text{or}}{+} bc^*)^*$$

L = set of all strings over $\{a, b, c\}$ that do not contain ac.

⇝ ◦ Proof

Integer constants K & R

octal $0\,(0 + 1 + \cdots + 7)^*$

hex $(0x + 0X)\,(0 + \cdots + 9 + A + \cdots + F + a + \cdots + f)^*$

→ may be suffixed by u or U, L or l;

Is the limitation set by language or the compiler?

→ Code compliance ⟶ lU works but l can be confused with 1.

→ LU ✓ lU ⇝ disallowed / warn by some compilers.

$$\begin{tikzcd}
\text{unsigned} \arrow[r] \arrow[dr] & \text{one more bit} \\
& {}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{unsigned} \arrow[r] \arrow[dr] & \text{type qualifier} \\
& \text{keyword}
\end{tikzcd}$$

```
unsigned int a = ~
int b = a;
```

```
64 — double aa;
32 — float b1 = aa;
```

$$\left.\begin{matrix} 4 \\ 8 \\ 16 \end{matrix}\right\} \text{Neural net}$$

$$\text{useless precision} \left\{\begin{matrix} 32 \\ 64 \\ 80 \\ 128 \end{matrix}\right.$$

LEX file .l $\rightsquigarrow$ lexical analyser rules

D [0-9]

letter ← L

hex ← H

exponent ← E

auto = $\underset{\sim}{\text{conditional}}$

char or long

↓

longest

auto ~ = ~~~~~~

↓

if this is compile type evaluation

→ auto keyword in C

→ auto in C++

"pythonification"

⤳ gcc and clang source code.

Explore source code if manuals don't work.

→ Backward compatibility is very important

– Intel.

→ auto in function argument ⤳ f (auto . ...)

⇣

OOPs will

→ Character constants octal \000 ⤳ ⁉

→ Floating Constant