COMP302 Lecture 9 30 September 2016

Scanner (1)

- chars into tokens *see 9c302*
- tokens:
  - VAR WS ID WS EQ WS NUM SC WS FOR WS LB …
- values
  - var ‘ ‘ f ‘ ‘ = ‘ ‘ 1 ; ‘ ‘ for ‘ ‘ (
- Sometimes, token = value, but that is not always the case.
- We could prune out irrelevant stuff – whitespace, comments.
- We need to build a scanner to identify each one of these symbolic pieces.
- Identify individual tokens using regular expressions.
- We feed our chars to the scanner, it identifies regexes, and assigns tokens to each.
- In many actual compilers, regexes are convert into finite automata [(non-) deterministic]
- In practice however, we will use regexes directly.
- We will need a regex for each token.

Token	Regex
;	/^;/
‘var’	/^var/
identifier	/^[a-zA-Z_][a-zA-Z_0-9]*

Exercise: numbers?

- Integers easy
- Floating points: harder because many ways to write the same things (0.3 ; .3; 3.0; 3.)

Notice that some regexes overlap. e.g. regex for var and regex for identifier both match the string “var”.
e.g. /^=/, /^==/, /^===/. In the last case, do I mean one assignment and one loose equality? Do I mean a strong equality…?

We will prioritize regexes

- e.g. /^var/ > /^[a-z…]
- longest matches. === → strict equality rather than three = in a row.
- context is important. “These are the tokens we expect”. Might only look for some subset of tokens at a given time. *see 9c302-1*
- Often have a default token
  - token : T_DEFAULT
  - value : str[0]
- Might want to look for the end of the string
  - token : T_EOF (end of file)

Chars + set of expected tokens → Scanner → Tokens.

Parsing and grammars (2)

- Build higher level structure from the tokens
- Specify the structure with a grammar
  - Ex: What is a program? A bunch of declarations.
    - Some for variables
    - Some for functions
  - Fundecl -> type identifier '(' arglist ')'
  - LHS -> RHS.
  - We create rules where the LHS is 'non-terminal' and the RHS is a list of terminals (tokens) or non-terminal (rule LHS).
  - Only one non-terminal on LHS → context-free grammar.

int foo(int x, int y) { } statement

Syntax 'BNF': Backus-Naur Form.

- Fundecl ::=. Sometimes non-terminals are in < >.
- <Fundecl> ::= <type> ID …
- <arglist> ::= <arg> ',' <arglist> | ε

ε is the empty symbol

- Recursion is the key idea, and we will make a lot of use of the empty symbol