**CS 131 Programming Languages**

**October 6, 2009**

*Combine Choosers*

You're given two functions, never look inside them, but build a scaffold around them. Let's try to write a function that takes three choosers and lets the first chooser pick which of the other two choosers to use.

```
let ccc c1 c2 c3 = fun (s1, s2, s3) n ->
        let (t1, r1) = (c1 s1 2) in
                if r1 = 0
                then let (t2, r2) = (c2 s2 n) in ((t1, t2, s3), r2)
                else let (t3, r3) = (c3 s3 n) in ((t1, s2, t3), r3) ;;
```

*Homework 2*

This assignment is about gluing stuff together in the bigger functions, such as combining two choosers. Generate and test: write a program that generates a parser. Generator is the matcher and tester is the receiver. Matcher is given a string to parse – there is multiple ways to parse the string. The acceptor may accept or reject the result. When acceptor rejects, the program backtracks to generate another result. When acceptor accepts, the program completes successfully. If matcher runs out of tries, the program fails. Matcher is a function that takes a fragment and an acceptor, returning 'x option (returns whatever acceptor returns):

```
matcher:
        frag -> acceptor -> 'x option
```

The acceptor takes a rule list and a fragment and returns 'x option (some value that gives further information about what it accepted):

```
acceptor:

        rule list -> frag -> 'x option
```

Therefore:

```
matcher:

        frag -> (rule list -> frag -> 'x option) -> 'x option
```

The 'x type: 'x = rule list * frag. Therefore:

```
matcher:

        frag -> (rule list -> frag -> (rule list * frag)) -> (rule list * frag)
```

*Lecture*

*Theory*

What can go wrong with the grammar?

- Ambiguity
- Symbols used but not defined
- Symbols defined but not used
- Useless production rules
- Looping (special case of useless rule)

*Practice*

- Extra constraints that the grammar cannot capture
    a. `int id (int i) { return n; }`
    b. *n* is used but not defined, yet it is a perfectly valid grammar
- It is possible to construct grammar that captures enough details to only allow correct language

- Such grammar may be too complicated / have too much detail for people to read and understand

*Grammar 1 (ambiguous, abstract parse/syntax tree):*

- `E -> E + E`
- `E -> E * E`
- `E -> ID`
- `E -> (E)`

*Grammar 2 (too much detail, concrete parse/syntax tree):*

- `E -> E + T`
- `E -> T`
- `T -> T * F`
- `T -> F`
- `F -> (E)`
- `F -> ID`

People use diagrams as a shorthand for BNF and EBNF.

*A really short C program*

```
# include <stdio.h>
int main (void) { return !getchar(); }
```

The C compiler's steps in understanding the file:

- Phase 1 (lexing):
    - tokenization | lexing / lexical analysis, results in a series of tokens that do not contain the names of the literals
    - combination of the tokens and source code is called lexeme
- Phase 2 (parsing):
    - creation of the parse tree
    - visiting the leafs of the parse tree right-to-left results in the original lexeme
- Type checking: definitions of all identifiers, duplicate definitions, common errors
    - generates intermediate code (not a real machine code):
        - main:
            - push getchar
            - call 0
            - not
            - return
    - the code is for a stack machine
- Assembly generation (*.s):
    - main:
        - call getchar
        - tstl r0
        - movcw 2,r0
        - ret
- Object code generation (by assembler)
    - machine code
    - symbol table, containing addresses of all functions called (some addresses unknown)
- Linking
    - actual pure machine code that can be executed
- Loading
    - copy the program into RAM and set IP/PC into the first instruction in the program

*Approaches to Generating a Program*

- Method A (seen): software tools (tokenizer, parser, assembler, linker, loader). Unix.
    - Portable, scalable
- Method B: integrated development environment (IDE), one big program to do all steps.