15-411: Compiler Design — Fall 2017

Recitation 3: Parsing — 22 September

The Lexer and Parser Generators

For Lab 2, you will need to update your lexer and parser to handle the new symbols and language features. We will look at parsing a `while` loop. First, we update the lexer. In OCaml, we change the file `c0Lexer.mll`. We currently have a block of definitions like

```
1 rule initial =
2  parse
3    ws+      { initial lexbuf }
4  | '\n'     { ParseState.newline (start lexbuf); initial lexbuf }
5  ...
6  | "if"     { assert false }
7  | "else"   { assert false }
8  | "while"  { assert false }
9  | "for"    { assert false }
```

We need to define new tokens for our keywords. Once we do, our lexer definition will look something like

```
1 rule initial =
2  parse
3    ws+      { initial lexbuf }
4  | '\n'     { ParseState.newline (start lexbuf); initial lexbuf }
5  ...
6  | "if"     { T.IF }
7  | "else"   { T.ELSE }
8  | "while"  { T.WHILE }
9  | "for"    { T.FOR }
```

Now we need to update our parser. Here you have more choices to make, but it should roughly look like the language definition in the Lab 2 handout.

```
1  stmt :
2      decl SEMI                      { marks (A.Declare $1$) (1, 1) }
3    | simp SEMI                      { marks $1 (1, 1) }$
4    | RETURN exp SEMI                { marks (A.Return $2) (1, 2) }$
5    | WHILE LPAREN exp RPAREN stmt   { marks (A.While ($3, $5) (1, 4) }
6      ;
```

Shift-Reduce Conflicts

A shift-reduce conflict occurs when it is ambiguous whether the parser should *shift* (save the current token on the stack and read another) or *reduce* (apply one of the rules from the grammar to the current token stack).

Checkpoint 0

Show that the following grammar has a shift-reduce conflict by showing two different ways to parse the string `200 * 2 + 11`.

$E \to E + E$

$E \to E * E$

$E \to [0-9]*$

$E \to (E)$

Reduce-Reduce Conflicts

A reduce-reduce conflict occurs when more than one rule in the grammar applies to the current token stack.

Checkpoint 1

Show that the following grammar has a reduce-reduce conflict by showing a successful and an unsuccessful parse of the string `bbbc`.

$S \to Cc$
$S \to Dd$
$C \to \epsilon$
$C \to Cb$
$D \to \epsilon$
$D \to Db$

Type Checking

Your L1 starter code includes a basic framework for a typechecker. You will need to make significant changes so that it can handle (non-trivial) blocks and Booleans. Type checking is very recursive in nature, so you'll want to have a function that recurses over the structure of your AST. Having immutable datatypes will make checking scope easy: if you pass your scope around recursively, you can simply modify the scope upon declaration and it will obey the scoping rules for blocks.

Checkpoint 2

Write the function prototype for a recursive function that you might use in your L2 compiler to typecheck an expression.

Grab Bag of Hints

- For the expression `if (a < 0) if (b < 0) x = 4 else x = 5`, x is not assigned if $a \geq 0$ (`else` binds to the most recent `if`)
- You have to add support for Boolean variables now, and you will have to add support for pointers in lab 4. Plan ahead when making design decisions to support different types in type checking and instruction selection.
- We suggest adding support for a `-O0` flag that disables register allocation and places all temps on the stack. Interference bugs fail in subtle, hard to understand ways.
- You can step through your programs with gdb. Place a breakpoint with `break _c0_main`, then use `step` to advance the program.