15-411: Compiler Design — Fall 2020

Recitation 4: Lexing & Parsing Solutions — 2 October

Before you can do any sort of analysis or transformations in your compiler, you have to perform **lexical analysis**. Lexical analysis transforms your program from a string of characters to a list of tokens, and assigns each token a grammatical meaning. From here, we **parse** those tokens to generate the abstract syntax tree.

Regular Expressions

First we need to define our alphabet. For L1, (and all of the other labs), we will be working with the ASCII character set. Once we have an alphabet, we define our tokens. Our definitions will be **regular expressions** (used by flex and ocamllex!).

Remember from the lecture notes:

Regular expressions r, s are expressions that are recursively built of the following form:

regex	matches
a	matches the specific character a from the input alphabet
$[a-z]$	matches a character in the specified range of letters a to z
ϵ	matches the empty string
$r\|s$	matches a string that matches r or one that matches s
rs	matches a string that can somehow be split into two parts, the first matching r, the second matching s
r^*	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 0$

Checkpoint 0

Define a regular expression that matches:

- `bool` type keyword
- A valid L1 Identifier
- A Hexadecimal

Solution:
Boolean Keyword: `bool`
Identifier: `[A-Za-z_][A-Za-z0-9_]*`
Hex constant: `0(X|x)[A-Fa-f0-9][A-Fa-f0-9]*`

Finite Automata

Regular expressions lend themselves naturally to finite automata. Implementation is mostly straightforward, and there are some fun proofs that say regular expressions and finite automata are interchangeable.

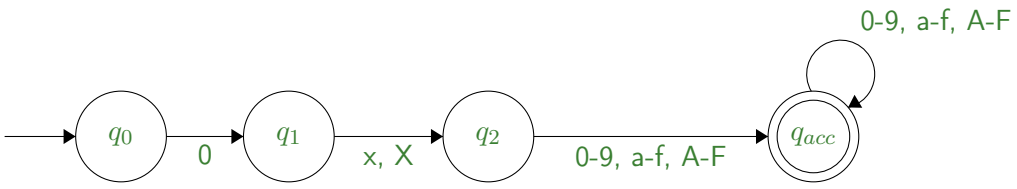
When constructing an automata for a regular expression, we need to be careful and accept the longest possible valid token. If we did not have that guarantee, `formula` might lex as the keyword `for` followed by an identifier, rather than as a single identifier.

In lecture, we discussed details of how we can efficiently convert a NFA into a DFA, but in your compiler you will use a lexer generator that will handle the implementation for you. You simply define tokens in a regular expression-like manner and the lexer will generate a function that consumes a string input and produces a list of tokens. Yet we still believe this to be a useful exercise.

Checkpoint 1

Create a DFA that accepts valid hexadecimal constants and rejects all other strings.

Solution:



Lexing

Checkpoint 2

Remember that the lexer is responsible for reading in an input program/string and producing a stream of tokens/symbols that are then later consumed by the parser. As an exercise, try lexing the following segment of a C0 program. You may choose whatever textual representation you deem best for each symbol (i.e `"("` $\rightarrow$ `"LPAREN"`).

```
1   if (score < 100) {
2      return 1;
3   }
```

Solution:

```
if  (       score  <         100       )       {       return  1         ;     }
IF  LPAREN  IDENT  LESSTHAN  DECCONST  RPAREN  LBRACE  RETURN  DECCONST  SEMI  RBRACE
```

Grammars & Parsing

Now once you have a stream of tokens from the lexer, the parser can now construct a parse tree from the stream of tokens. Recall from lecture a grammar G for a language L(G) is defined by a set of productions $\alpha \to \beta$ and a start symbol S, a distinguished non-terminal symbol.

For a given grammar G with start symbol S, a derivation in G is a sequence of rewritings $S \to \gamma_1 \to \to \gamma_n = w \in L(G)$ in which we apply productions from G. **Parsing** uses this derivation process to produce a parse tree (derivation) for w, in which the nodes represent the non-terminal symbols and the root being S.

We run into ambiguities when there are multiple possible parse trees for the same token stream. Below we'll take a closer look at possible ambiguities.

Grammar Ambiguities

Ambiguities can result as a consequence of the production rules and symbols chosen in defining a grammar G. An ambiguity in the grammar arises when there are multiple possible valid parse trees for the same token stream.

Checkpoint 3

Given the context-free grammar G containing productions of the form:

$$\begin{array}{lcll} \gamma_1 & : & A & \to & \textbf{A + A} \\ \gamma_2 & : & A & \to & \textbf{A - A} \\ \gamma_3 & : & A & \to & \text{int} \\ \gamma_4 & : & A & \to & \text{id} \end{array}$$

Prove that the grammar G is ambiguous by showing two parse trees for the stream $1 + 2 - \texttt{id}_\texttt{x}$.

Solution:
Parse Tree 1: $A \to A + A \to 1 + (A - A) \to 1 + (2 - A) \to 1 + (2 - id_x)$
Parse Tree 2: $A \to A - A \to (A + A) - A \to (1 + 2) - A \to (1 + 2) - id_x$

Conflicts in a LR(k) Parser

Now we will discuss shift-reduce and reduce-reduce conflicts common in LR(k) parsers. Remember from lecture that a bottom-up LR(k) parser parses from left-to-right in a single-pass with right-most derivation using k look-ahead tokens. A shift-reduce parser holds viable prefixes on a stack along with `k` lookahead symbols with the input stream containing remaining symbols.

LR(k) parsers at each step must determine whether the parser should *shift* or *reduce*. *Shifting* saves the current token on the maintained stack and reads another token while *reducing* applies some rule from the grammar to the front of the current token stack. As such, LR(k) parsers are prone to two common issues when dealing with certain grammars: **shift-reduce** and **reduce-reduce** conflicts.

Shift-Reduce Conflicts

A shift-reduce conflict occurs when it is ambiguous whether the parser should *shift* or *reduce*.

Checkpoint 4

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

$E \to E + E$

$E \to E * E$

$E \to [0-9]*$

$E \to (E)$

Solution:

```
          || 200 * 2 + 11
      200 || * 2 + 11
        E || * 2 + 11
      E * || 2 + 11
    E * 2 || + 11
    E * E || + 11
E || + 11   or   E * E + || 11
  ...                    ...
```

Reduce-Reduce Conflicts

A reduce-reduce conflict occurs when more than one rule in the grammar can be applied.

Checkpoint 5

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

$S \to Cc$

$S \to Dd$

$C \to \epsilon$

$C \to Cb$

$D \to \epsilon$

$D \to Db$

Solution:

```
          || bbbc
 C || bbbc  or  D || bbbc
Cb || bbc      Db || bbc
 C || bbc       D || bbc
Cb || bc       Db || bc
 C || bc        D || bc
Cb || c        Db || c
 C || c         D || c
Cc ||          Dc ||
 S ||             ??
```

Static Semanntics

Static semantics is a collection of rules for determining whether a program is well-formed or not. Static semantics does not address the issue of runtime behavior but instead answers questions of the nature: are all expressions and statements well-formed, are variable usages correct (i.e declare before use for L2), and do all functions end with an appropriate return statement – amongst other questions based off of the AST.

Type Checking

Type checking addresses the question of whether all expressions and statements are well-formed when checked against the type system and its rules. For instance, under the L2 type system and L2 rules, type checking is used to ensure that for a statement `int x = y`, the variable `y` is an `int` and not a `bool`.

Type checking rules are typically written down as inference rules. An example inference rule for type checking a relational operation is denoted below.

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int} \quad \texttt{relop} \in \{==, >=, <=, <, >\}}{e_1 \ \texttt{relop} \ e_2 : \texttt{bool}}$$

For those unfamiliar with reading inference rules, the inference rule contains three premises: (1) e_1 is an `int` type, (2) e_2 is an `int` type, and (3) `relop` is one of `{==, >=, <=, <, >}`. If the premises are satisfied, one can then conclude that e_1 `relop` e_2 is valid and has type `bool`. If no inference rule can be applied to judge an expression or statement valid, then the type checker deems the expression or statement invalid.

Checkpoint 6 Writing a type checker

Your L1 starter code includes a basic framework for a type checker. You will need to make changes so that it can handle (non-trivial) blocks and booleans. Type checking is recursive in nature so you will likely have some function that recurses over the AST.

Consider the following set of guiding questions:

- What type-related metadata needs to be stored?
- How will the metadata be stored and accessed for each variable?
- How will type metadata for variables be maintained across scopes?
- What are the inputs to the type checker?
- How will the type checker be invoked?
- How will the type checker check expressions and statements?

Solution: These are just guiding questions. There is no "correct" solution.

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 and other types in lab 4. Plan ahead when making design decisions to support different types in type checking, instruction selection, and x86 code emission. We advise you to design a systematic approach for annotating and querying type-related metadata for variables and handling scoping.
- 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.