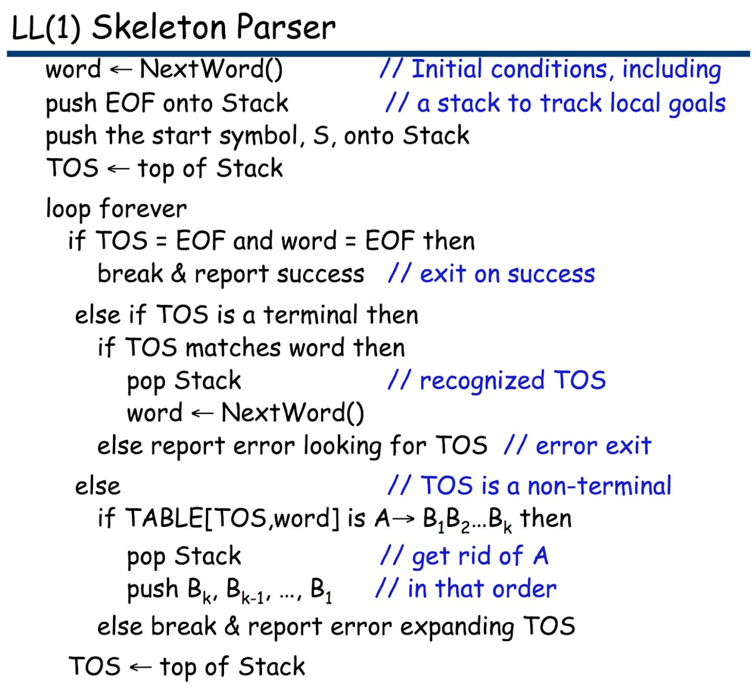
Parser Design

Christopher Tam
Dhruv Mehra

Architecture of the Parser

The parser implementation is in its own parser Rust submodule, and it contains all the necessary functions and definitions required to parse a Tiger program. Specifically we chose to implement LL(1) parser for this assignment because we were most familiar with the underlying concepts required to build a LL(1) parser.

The parser implements the pseudocode mentioned in the slides:



LL(1) Skeleton Parser

```
word ← NextWord()                // Initial conditions, including
push EOF onto Stack              // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and word = EOF then
    break & report success     // exit on success
  else if TOS is a terminal then
    if TOS matches word then
      pop Stack                // recognized TOS
      word ← NextWord()
    else report error looking for TOS  // error exit
  else                         // TOS is a non-terminal
    if TABLE[TOS,word] is $A \rightarrow B_1 B_2 \ldots B_k$ then
      pop Stack                // get rid of A
      push $B_k$, $B_{k-1}$, …, $B_1$      // in that order
    else break & report error expanding TOS
  TOS ← top of Stack
```

Generating the Grammar and Lookahead Table

First the Grammar provided in the Tiger specification was transcribed to a .txt file - see data/grammar.txt. Then Left recursion was removed from productions involving EXPR, TERM, AEXPR and CLAUSE. Further productions containing IDS, NEPARAMS, STMT, NEEXPRS, PRED, and FACTOR was left factored.

Once the grammar was written out, a script was written to convert this text grammar into a JSON object suitable for use by the program. The script can be found in the data directory and is called generator.py.

The grammar in grammar.txt was put into hackingoff.com's LL(1) table generator. This output a lookahead table as a JSON object which was then used by our parser.

Abstract Syntax Tree

We just use a stack to keep track of the start of any production that **used** to be a left recursive production, and every time its prime (') production produces a non-epsilon output, we insert the non-prime production at the place we kept track of.

If the prime (') production produces epsilon, we pop off the stack. To remove the left-factoring introduced we just removed the prime productions to go back to the original syntax tree.

Implementation Details

2 new structures were defined specifically to parse the json objects:

```
pub struct Grammar {
   pub nonterminals: Vec<String>,
   pub productions: Vec<Vec<String>>
}

pub struct ParseTable {
   pub terminals: Vec<String>,
   pub table: Vec<Vec<usize>>
}
```

The nonterminal field inside Grammar is a vector of strings where each entry is a nonterminal in the grammar G.

The field productions is a 2D vector of Strings where the index of the production is the production number for that rule, and each entry in a production is an integer or a word. An integer implies a non-terminal which can be indexed into the nonterminals vector and a word implies a terminal token.

Similar to nonterminals, the terminals field in ParseTable struct is a vector of non-terminals present in the grammar G.

The table field in the ParseTable is the actual lookahead table, where each entry table[index of nonterminal][index of terminal] is index of the production that the nonterminal expands to.

Challenges

Debugging the parse table was particularly hard especially when there were some non-terminals which hadn’t been left refactored or were left recursive.

Converting the new modified AST T’ back original AST T was especially hard, and required a lot of debugging.