SI 413: To err is human
to really #@&% up requires a computer

Professor Keith Sullivan

Recall Compilation

- Scanning: Turning source code into a token stream. This stage removes white space and comments, and identifies what kind of token each piece of the code is.
- Parsing: Turning a token stream into a parse tree. This stage checks that the sequence of tokens is grammatically correct and can be grouped together according to the specifications of how the language works.
- Semantic analysis: Turning a parse tree into an abstract syntax tree. This stage cuts out any unnecessary or redundant information from the parse tree to form a concise representation of what the program means (i.e., the AST).
- Code generation: Turning that AST into executable machine code.

Programming Language Specification

- Programming languages provide a medium to describe an algorithm so a computer can understand it.
- **How do we describe** such a programming language?
- Need to specify:
  - Syntax: rules for how a program looks
  - Semantics: the *meaning* of a program

C++ Example

```
int x = 2;
x = 2^3;
```

```
if (x < y < z)
    return y;
else
    return 0;
```

Representation

- How to represent (describe) syntax and semantics so a computer can understand it?
- Tokens: are regular expressions represented as finite automoton (FA).
- Semantics: merge multiple tokens via a context free grammar, represented as a push down automaton (PDA)

Simple Calculator Example

The tokens for a simple calculator:

$$\begin{aligned} OP &= + \mid - \mid * \mid / \\ NUM &= \text{"}-\text{"}?[0-9]+ \\ STOP &= ; \end{aligned}$$

and the associated grammar:

$$\begin{aligned} S &\rightarrow exp\ STOP \\ exp &\rightarrow exp\ OP\ exp \mid NUM \end{aligned}$$

What is wrong with this grammar?

Better Grammar

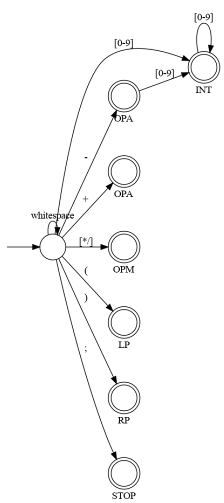
More tokens

$$
\begin{aligned}
OPA &= [+-]\\
OPM &= [*/]\\
NUM &= "-"?[0-9]+\\
LP &= "("\\
RP &= ")"\\
STOP &= ";"
\end{aligned}
$$

and the associated non-ambiguous grammar

$$
\begin{aligned}
S &\rightarrow exp\ STOP\\
exp &\rightarrow exp\ OPA\ term \mid term\\
term &\rightarrow term\ OPM\ factor \mid factor\\
factor &\rightarrow NUM \mid LP\ exp\ RP
\end{aligned}
$$

Hand Rolled FA for Calculator



Tokens

- What is a token?
- Return the terminal symbol
- Some tokens require additional information

Look-ahead in Scanners

- How to know when a token ends?
- How many tokens is 123*-54 using our scanner?
- Maximal munch:
- Naive approach is $O(n^2)$
- How to get back to $O(n)$?

Parsing

- Parsing is the second part of syntax analysis.
- Grammars specify how to combine tokens using a parse tree with tokens as the leaves.
- Unlike theory class, we want fast grammars.

Generalize or Specialize?

- Parsing a CFG deterministically is hard! General case is $O(n^3)$.
- But, if we restrict the class of CFGs, we can parse much faster.
- We want $O(n)$ using a single stack and not too much look-ahead.

Parsing Strategies

Top Down

- Constructs parse tree starting at the root
- "Follow the arrows" – carry production rules forward.
- Requires predicting which rule to apply for a given non-terminal.
- LL: Left-to-right, leftmost derivation.

Bottom Up

- Constructs parse tree starting at the leaves
- "Go against the flow" – apply reduction rules backwards
- LR: Left-to-right, rightmost derivation

Top Down Parsing

Initialize the stack with S, the start symbol.;
while *stack and input are both not empty* **do**
    if *top of stack is a terminal* **then**
        Match terminal to next token
    end
    else
        Pop nonterminal and replace with r.h.s. from a derivation rule
    end
end
Accept iff stack and input are both empty

Example

Recall calculator grammar:

$$
\begin{aligned}
S &\rightarrow exp\ STOP \\
exp &\rightarrow exp\ OPA\ term \mid term \\
term &\rightarrow term\ OPM\ factor \mid factor \\
factor &\rightarrow NUM \mid LP\ exp\ RP
\end{aligned}
$$

Parse $3 + 4 * (20/5)$ both top-down and bottom up

LL(1) Grammars

A grammar is LL(1) if it can be parsed with just 1 token's worth of look-ahead.

Example grammar

$$\begin{array}{lcl} S & \to & X\ X \\ X & \to & a\ \ b \\ & \to & a\ \ a \end{array}$$

Is this LL(1)? Why or why not?

Common Prefixes

The common prefix in the previous grammar causes a problem.

Can "factor out" the common prefix.

$$\begin{array}{lcl} S & \to & T\ T \\ T & \to & a\ X \\ X & \to & a \\ & \to & b \end{array}$$

Left Recursion

The other enemy of LL(1) is left recursion.

$$\begin{array}{lcl} S & \to & exp \\ exp & \to & exp\ +\ NUM \\ & \to & NUM \end{array}$$

Why isn't this LL(1)?

How can we fix it?

Tail rules to get LL

To make LL grammars, we typically add extra tail rules for list-like non-terminals.

For instance:

$$
\begin{array}{lll}
S & \rightarrow & exp \\
exp & \rightarrow & NUM\ exptail \\
exptail & \rightarrow & \epsilon \\
 & \rightarrow & +\ NUM\ exptail
\end{array}
$$

Dangling Else

Consider the following grammar from Pascal:

$$
\begin{array}{lll}
stmt & \rightarrow & IF\ condition\ then\_clause\ else\_c \\
 & \rightarrow & other\_stmt \\
then\_clause & \rightarrow & THEN\ stmt \\
else\_clause & \rightarrow & ELSE\ stmt \\
 & \rightarrow & \epsilon.
\end{array}
$$

How is the following code parsed?

```
if C1 then
if C2 then
S1
else
S2
```

Solution

$$
\begin{array}{lll}
stmt & \rightarrow & balanced\_stmt \\
 & \rightarrow & unbalanced\_stmt \\
 & & \\
balanced\_stmt & \rightarrow & IF\ condition\ THEN\ balanced\_stmt \\
 & & \qquad ELSE\ balanced\_stmt \\
 & \rightarrow & other\_stmt \\
 & & \\
unbalanced\_stmt & \rightarrow & IF\ condition\ THEN\ stmt \\
 & \rightarrow & IF\ condition\ THEN\ balanced\_stmt \\
 & & \qquad ELSE\ unbalanced\_stmt
\end{array}
$$

Follow and Predict Sets

PREDICT
The PREDICT set of any production rule for a nonterminal contains any token which could come first in parsing that nonterminal, or in case of an epsilon production, anything which could come immediately afterwards.

FOLLOW
The FOLLOW set of any nonterminal consists of all the tokens which might come immediately after that nonterminal in a parse.

Bottom up Parsing

A bottom-up (LR) parser reads tokens from left to right and maintains a stack of terminal *and* non-terminal symbols.

At each step it does one of two things:

- Shift: Read in the next token and push it onto the stack
- Reduce: Recognize that the top of the stack is the r.h.s. of a production rule, and replace that r.h.s. by the l.h.s., which will be a non-terminal symbol.

The question is how to build an LR parser that applies these rules *systematically*, *deterministically*, and of course *quickly*.

Example LR grammar

$$\begin{array}{lcl} S & \to & E \\ E & \to & E + T \\ & \to & T \\ T & \to & n \end{array}$$

What is bottom up parse of $n + n$?

How do we maintain "state" of the parser?

Parser States

At any point during parsing, we are trying to expand one or more production rules.

The state of a given (potential) expansion is represented by an "LR item".

For our example grammar we have the following LR items:

$$\begin{array}{lll}
S & \to & \bullet\ E \\
S & \to & E\ \bullet \\
E & \to & \bullet\ E + T \\
E & \to & E\ \bullet + T \\
E & \to & E + \bullet\ T \\
E & \to & E + T\ \bullet \\
E & \to & \bullet\ T \\
E & \to & T\ \bullet \\
T & \to & \bullet\ n \\
T & \to & n\ \bullet
\end{array}$$

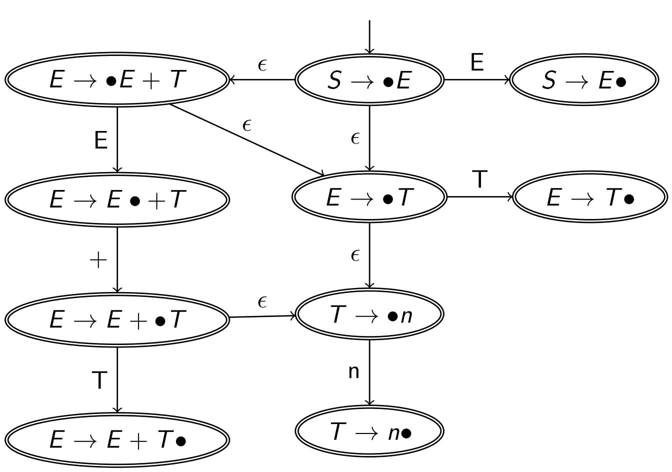
Characteristic Finite State Machine

The CSFM (Characteristic Finite State Machine) is a FA representing the transitions between the LR item "states".

There are two types of transitions:

- Shift: consume a terminal or non-terminal symbol and move the $\bullet$ to the right by one.
  Example: $(T \to \bullet n) \xrightarrow{\text{n}} (T \to n\bullet)$
- Closure: If the $\bullet$ is to the left of a non-terminal, we have an ϵ-transition to any production of that non-terminal with the $\bullet$ all the way to the left.
  Example: $(E \to E + \bullet T) \xrightarrow{\epsilon} (T \to \bullet n)$

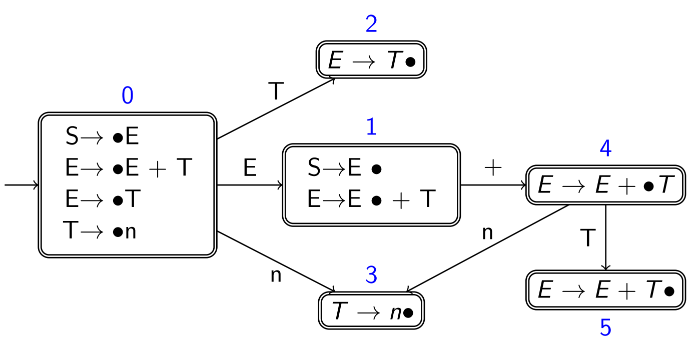
Nondeterministic CFSM



CFSM Properties

- Observe that every state is accepting.
- This is an NDFA that accepts *valid stack contents*.
- The "trap states" correspond to a reduce operation: Replace r.h.s. on stack with the l.h.s. non-terminal.
- We can simulate an LR parse by following the CFSM on the current stack symbols AND un-parsed tokens, then starting over after every reduce operation changes the stack.
- We can turn this into a DFA just by combining states.

Deterministic CFSM



- Every state is labelled with a number.
- Labels are pushed on the stack along with symbols.
- After a reduce, go back to the state label left at the top of the stack.

SLR

Parsing this way using a (deterministic) CFSM is called *SLR Parsing*.

Following an edge in the CFSM means shifting;
coming to a rule that ends in • means reducing.

SLR(k) means SLR with k tokens of look-ahead.
The previous grammar was SLR(0); i.e., no look-ahead required.

When might we need look-ahead?

SLR Conflicts

A conflict means we don't know what to do!

- Shift-reduce conflict:

  $\boxed{\begin{array}{l} W \to a \bullet \\ W \to a \bullet b \end{array}}$

- Reduce-reduce conflict:

  $\boxed{\begin{array}{l} W \to a \bullet \\ X \to a \bullet \end{array}}$

SLR(1)

SLR(1) parsers handle conflicts by using one token of look-ahead:

- If the next token is an outgoing edge label of that state, shift and move on.
- If the next token is in the FOLLOW set of a non-terminal *that we can reduce to*, then do that reduction.

Of course, there may still be conflicts, in which case the grammar is not SLR(1). More look-ahead may be needed.

Problem Grammar 1

Draw the CFSM for this grammar:

$$\begin{array}{rcl} S & \to & W\,W \\ W & \to & a \\ & \to & a\,b \end{array}$$

Problem Grammar 2

Draw the CFSM for this grammar:

$$
\begin{aligned}
S &\rightarrow W\ b \\
W &\rightarrow a \\
&\rightarrow X\ a \\
X &\rightarrow a
\end{aligned}
$$