Programming Language Specification

Programming languages provide a medium to describe an algorithm so that a computer can understand it.

But how can we **describe a programming language** so that a computer can understand it?

We need to specify both:

- Syntax: the rules for how a program can look
- Semantics: the *meaning* of syntactically valid programs

English Syntax vs. Semantics

Consider four English sentences:

- Burens mandneout exhastrity churerous handlockies audiverall.
- Feels under longingly shooting the darted about.
- Colorless green ideas sleep furiously.
  (Noam Chomsky)
- It's like all the big stories were stitched together into dead tiny sisters.
  (Jeffrey Harrison)

C++ Syntax vs. Semantics

What do the following code fragments mean?

- ```
  int x;
  x = 2^3;
  ```
- ```
  if (x < y < z) {
    return y;
  }
  else return 0;
  ```

Syntax feeds semantics!

Consider the following grammar:

exp → *exp op exp* | `NUM`
op → `+` | `-` | `*` | `/`

This correctly defines the syntax of basic arithmetic statements with numbers. But it is *ambiguous* and confuses the semantics!

Better syntax specification

Here is an unambiguous syntax for basic arithmetic:

Terminals (i.e., *tokens*)

```
 OPA = [+-]
 OPM = [*/]
 NUM = ("-"|)[0-9]+
  LP = "("
  RP = ")"
STOP = ";"
```

Valid constructs (i.e., *grammar*)

S → *exp* `STOP`
exp → *exp* `OPA` *term* | *term*
term → *term* `OPM` *factor* | *factor*
factor → `NUM` | `LP` *exp* `RP`

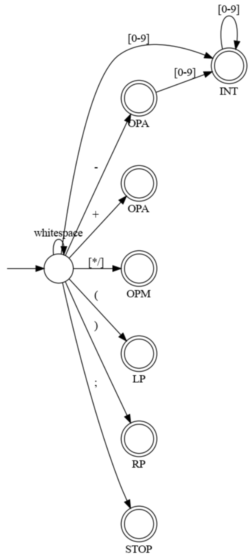
Scanner and Parser Specification

Recall that compilation begins with *scanning* and *parsing*.

- Scanning turns a raw character stream into a stream of tokens. Tokens are specified using *regular expressions*.
- Parsing finds larger syntactic constructs and turns a token stream into a parse tree. Grammar is specified in *Extended Backus-Nauer Form*. (EBNF allows the normal constructs plus Kleene +, Kleene *, and parentheses.)

Hand-rolled Scanner FA

Here is a finite automaton for our basic tokens:



What is a token?

When our FA accepts, we have a valid token.

We return the terminal symbol or "type".
This usually comes right from the accepting state number.

Some tokens may require additional information, such as the value of the number, or which operation was seen.

Code for hand-rolled scanner

The `calc-scanner.cpp` file implements the FA above using `switch` statements. Check it out!

There is also a Bison parser in `calc-parser.ypp` containing:

- Datatype definition for the "extra" information returned with a token
- Grammar production rules, using token names as terminals
- A main method to parse from standard in

Extending our syntax

Some questions:

- What if we wanted ** to mean exponentiation?
- How about allowing comments? Single- or multi-line?
- How about strings delimited with "?
- What about escape sequences?
- Can we allow negative and/or decimal numbers?

Maximal munch

How does the C++ scanner know that "/*" starts a comment, and is not a divide and then a multiply operator?

How does it know that "-5" is a single integer literal, and not the negation operator followed by the number 5?

How does it even know if "51" is two integers or one?

Looking ahead

The code we referenced uses `cin.putback()` to return unneeded characters to the input stream.

But this only works for a single character. In general, we need to use a buffer. Implementing this requires a circular, dynamically-sized array, and is a bit tricky.

For example, consider the language with - and --> as valid tokens, but not --. This requires 2 characters of "look-ahead".

Structure of a Scanner

How does a scanner generation tool like `flex` actually work?

1. An NDFA is generated from each regular expression.
   Final states are marked according to which rule is used.
2. These NDFAs are combined into a single NDFA.
3. The big NDFA is converted into a DFA. *How are final states marked?*
4. The final DFA is minimized for efficiency.
   The DFA is usually represented in code with a *state-character array*.

Look-ahead in scanners

The "maximal munch" rule says to always return the longest possible token.

But how can the DFA tell if it has the maximal munch?

Usually, just stop at a transition from accepting to non-accepting state.
This requires one character of *look-ahead*.

Is this good enough for any set of tokens?

Parsing

Parsing is the second part of syntax analysis.

We use grammars to specify *how tokens can combine*.
A parser uses the grammar to construct a parse tree
with tokens at the leaves.

Scanner: Specified with regular expressions, generates a DFA
Parser: Specified with context-free grammar, generates a . . .

Generalize or Specialize?

Parsing a CFG *deterministically* is **hard**:
requires lots of computing time and space.

By (somewhat) restricting the class of CFGs, we can parse much faster.

For a program consisting of n tokens, we want $O(n)$ time,
using a single stack, and not too much look-ahead.

Parsing Strategies

Top-Down Parsing:

- Constructs parse tree starting at the root
- "Follow the arrows" — carry production rules forward
- Requires *predicting* which rule to apply for a given nonterminal.
- LL: **L**eft-to-right, **L**eftmost derivation

Bottom-Up Parsing:

- Constructs parse tree starting at the leaves
- "Go against the flow" — apply reduction rules *backwards*
- Requires
- LR: **L**eft-to-right, **R**ightmost defivation

Parsing example

Simple grammar

$S \to T\ T$
$T \to \texttt{aa}$
$T \to \texttt{bb}$

Parse the string `aabb`, top-down and bottom-up.

Handling Errors

How do scanning errors occur?
How can we handle them?

How do parsing errors occur?
How can we handle them?

“Real” scanners/parsers also tag *everything* with filename & line number to give programmers extra help.

Top-down parsing

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

Make choice on Step 6 by “peeking” ahead in the token stream.

LL(1) Grammars

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

Example grammar

$S \to T\ T$
$T \to \texttt{ab}$
$T \to \texttt{aa}$

Is this grammar LL(1)?

Common prefixes

The *common prefix* in the previous grammar causes a problem.

In this case, we can "factor out" the prefix:

LL(1) Grammar

$S \to T\ T$
$T \to \texttt{a}\ X$
$X \to \texttt{b}$
$X \to \texttt{a}$

Left recursion

The other enemy of LL(1) is *left recursion*:

$S \to exp$
$exp \to \texttt{exp + NUM}$
$exp \to \texttt{NUM}$

- Why isn't this LL(1)?
- How could we "fix" it?

Tail rules to get LL

To make LL grammars, we usually end up adding extra "tail rules" for list-like non-terminals.

For instance, the previous grammar can be rewritten as

$S \to exp$
$exp \to \texttt{NUM}\ exptail$
$exptail \to \epsilon \mid \texttt{+ NUM}\ exptail$

This is now LL(1).

(Remember that ϵ is the empty string in this class.)

Recall: Calculator language scanner

Token name	Regular expression
NUM	(-|)[0-9]+
OPA	[+-]
OPM	[*/]
LP	"("
RP	")"
STOP	;

LL(1) grammar for calculator language

$$S \rightarrow exp\ \texttt{STOP}$$

$$exp \rightarrow term\ exptail$$
$$exptail \rightarrow \epsilon \mid \texttt{OPA}\ term\ exptail$$

$$term \rightarrow factor\ termtail$$
$$termtail \rightarrow \epsilon \mid \texttt{OPM}\ factor\ termtail$$

$$factor \rightarrow \texttt{NUM} \mid \texttt{LP}\ exp\ \texttt{RP}$$

How do we know this is LL(1)?

Recursive Descent Parsers

A recursive descent top-down parser uses *recursive functions* for parsing every non-terminal, and uses the function call stack implicitly instead of an explicit stack of terminals and non-terminals.

If we also want the parser to *do something*, then these recursive functions will return values. They will also sometimes take values as parameters.

(See posted example.)

Table-driven parsing

Auto-generated top-down parsers are usually *table-driven*.

The program stores an *explicit* stack of expected symbols, and applies rules using a nonterminal-token table.

Using the expected non-terminal and the next token, the table tells which production rule in the grammar to apply.

Applying a production rule means pushing some symbols on the stack.

(See posted example.)

Automatic top-down parser generation

In table-driven parsing, the code is always the same;
only the table is different depending on the language.

Top-down parser generators first generate two sets for each non-terminal:

- PREDICT: Which tokens can appear when we're expecting this non-terminal
- FOLLOW: Which non-terminals can come after this non-terminal

There are simple rules for generating PREDICT and FOLLOW, and then for generating the parsing table using these sets.

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*.

Simple grammar for LR parsing

Consider the following example grammar:

$$\begin{array}{l} S \to E \\ E \to E \texttt{ + } T \\ E \to T \\ T \to \texttt{n} \end{array}$$

Examine a bottom-up parse for the string `n + n`.

How can we model the "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:

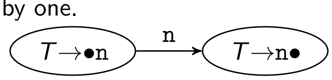
$$\begin{array}{ll} S \to \bullet\, E & E \to E \texttt{ + } T\, \bullet \\ S \to E\, \bullet & E \to \bullet\, T \\ E \to \bullet\, E \texttt{ + } T & E \to T\, \bullet \\ E \to E\, \bullet \texttt{ + } T & T \to \bullet\, \texttt{n} \\ E \to E \texttt{ + } \bullet\, T & T \to \texttt{n}\, \bullet \end{array}$$

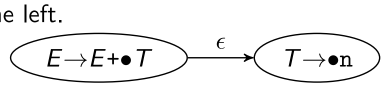
The • represents "where we are" in expanding that production.

Pieces of the CFSM

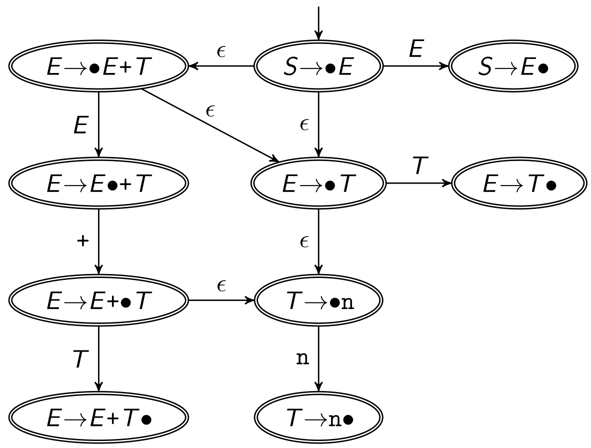
The CSFM (Characteristic Finite State Machine) is a FA representing the *transitions* between the LR item "states".

There are two types of transitions:

- **Shift**: consume a terminal *or non-terminal* symbol and move the • to the right by one.
  Example: 

- **Closure**: If the • is to the left of a non-terminal, we have an ϵ-transition to any production of that non-terminal with the • all the way to the left.
  Example: 

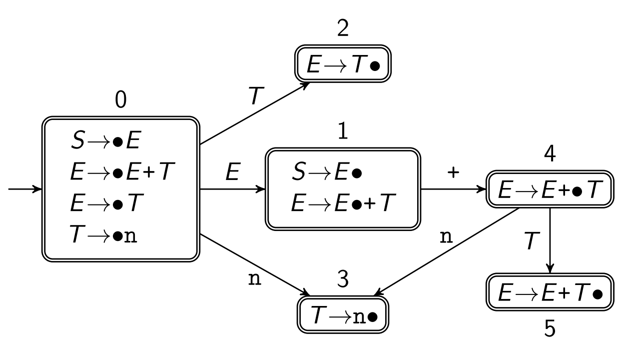
Nondeterministic CFSM



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?

Problem Grammar 1

Draw the CFSM for this grammar:

$S \to W\ W$
$W \to \texttt{a}$
$W \to \texttt{ab}$

Problem Grammar 2

Draw the CFSM for this grammar:

$S \to W\ \texttt{b}$
$W \to \texttt{a}$
$W \to X\ \texttt{a}$
$X \to \texttt{a}$

SLR Conflicts

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

- **Shift-reduce conflict**:

  $W \to \mathtt{a}\bullet$
  $W \to \mathtt{a}\bullet\mathtt{b}$

- **Reduce-reduce conflict**:

  $W \to \mathtt{a}\bullet$
  $X \to \mathtt{a}\bullet$

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.

Review: Scanning

Scanning means turning source code into tokens.

Scanners . . .

- are implemented with FAs.
- are specified with regular expressions.
- use a look-ahead character to implement *maximal munch*
- can be generated automatically. This involves determinizing an NDFA and then minimizing the DFA.

Review: Top-Down Parsing

Parsing means turning a token stream into a parse tree.

Top-down parsers . . .

- generate the parse tree starting with the root
- can recognize LL grammars
- need to *predict* which grammar production to take
- use token(s) of look-ahead to make decisions
- can be implemented by intuitive recursive-descent parsers
- can also be implemented by table-driven parsers

Review: Bottom-Up Parsing

Parsing means turning a token stream into a parse tree.

Bottom-up parsers . . .

- generate the parse tree starting with the leaves
- can recognize LR grammars
- can recognize more languages than LL parsers
- need to resolve shift-reduce and reduce-reduce conflicts
- use token(s) of look-ahead to make decisions
- can be implemented using CFSMs
- are created by Bison