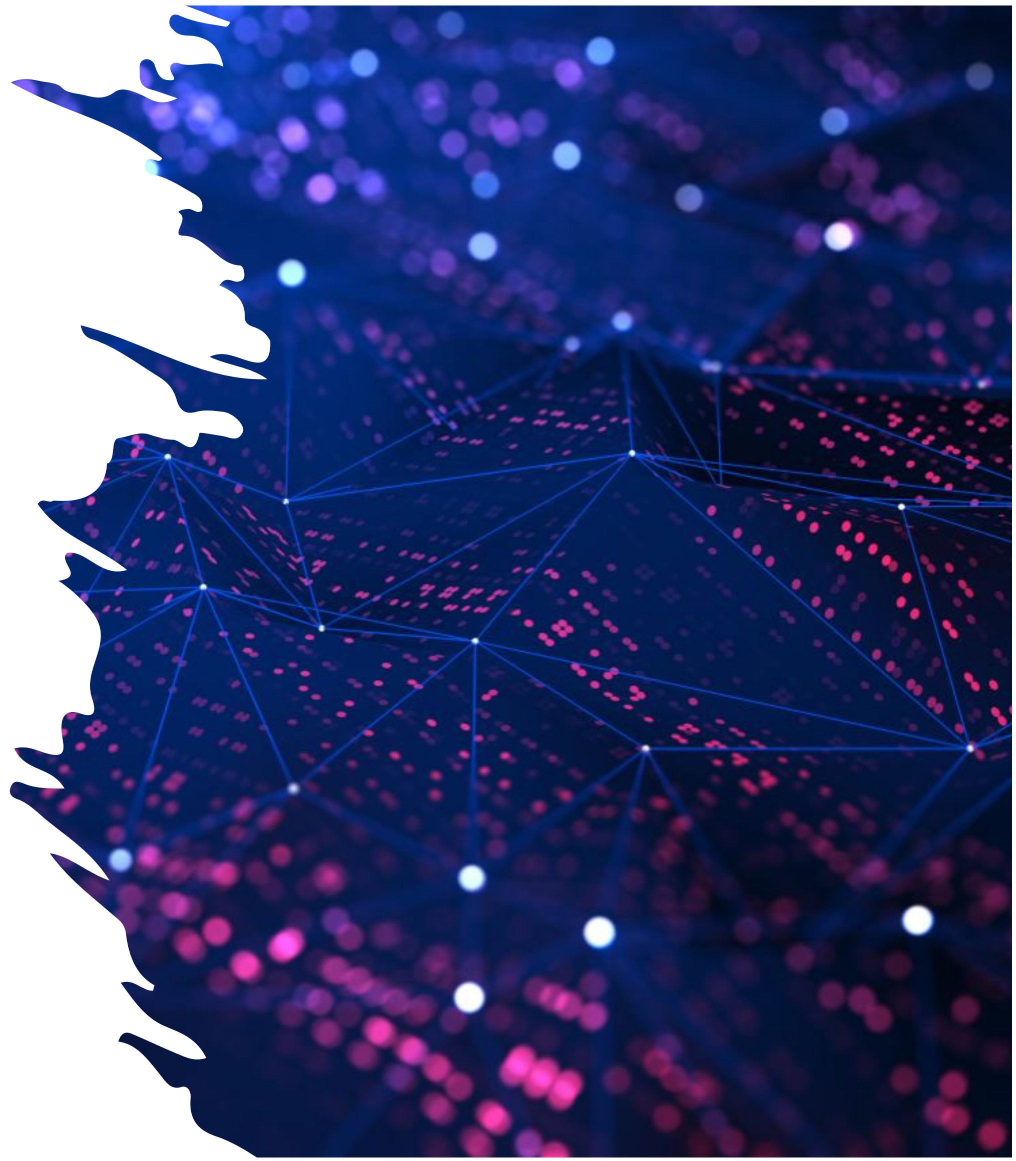
Building Parsers with DPDA

CSCI 3313 SPRING 2021



Grammar, Language, and Member Strings – 1

- A CFL: $\{ww^R \mid w \in \{a, b\}^*\}$
- Its CFG: $S \rightarrow aSa \mid bSb \mid \lambda$
- A string $z = aabbaa \in L$ generated through S:
  - $S \rightarrow aSa \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbaa$

- Another CFG
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \lambda$
  - $T \rightarrow FT'$
  - $T' \rightarrow * FT' \mid \lambda$
  - $F \rightarrow (\ E\ ) \mid \boldsymbol{id}$

- A string generated through E:
  - $E \rightarrow TE' \rightarrow FT' + TE'$
  - $\rightarrow \boldsymbol{id} * FT' + FT'\lambda$
  - $\rightarrow \boldsymbol{id} * \boldsymbol{id}\ \lambda + \boldsymbol{id}\ \lambda\lambda$
  - $\rightarrow \boldsymbol{id} * \boldsymbol{id} + \boldsymbol{id}$

Grammar, Language, and Member Strings – 2

- Yet another CFG
  - *program* → **prog id** ; *decls comp_stmt* **.**
  - *decls* → *decls* **var** *id_list* **:** *type* **;** | *λ*
  - *id_list* → **id** | *id_list* **, id**
  - *type* → **int** | **real**
  - *comp_stmt* → **begin** *stmt_list* **end**
  - *stmt_list* → … …

- Variables: internal nodes
- Terminals: leaf nodes

reserved words

variable* names

punctuations

numbers

operators

- A string/program generated through *program*:

prog main;

    var two, three, temp**: int;**

begin

    two **=** 2; three **=** 3;

    temp **=** 11 ***** two **+** three;

end.

*: In the context of program, not grammar; identifiers in general.

Membership Problem & Parsing – 1

- How do we check whether a string is in a language, or generated by a grammar? [**Iteratively checking** v. **Parsing**]
- Or, within the programming languages context, how do IDEs check for syntax errors? [**Parsing**]

- Iteratively checking (brute-force): $O(|P|^n)$ for **each** program with length n.

- CYK Algorithm: one of the parsing algorithms, $O(n^3)$ for **each** program with length n.

Membership Problem & Parsing – 2

❖What if amounts of rules and length are of thousands?

- May not HALT*? (Will be covered later) – Time Complexity
- Check again each time program is updated.

*: Can't differentiate on whether "we are still actively checking" or maybe "we are in an infinite loop."

❖What if there's productions such as $A \rightarrow aB \mid aC$ **?

- Guessing? Need to create multiple paths. How many? – Space Complexity
- Back tracking – Time Complexity
- Both could be of exponential orders

**: Not a DPDA; more than one productions for one ID.

✓There are better parsing algorithms!

- *LL(k)* Parsing
- *LR(k)* Parsing

Look-ahead Parsing – 1

- *LL(k)*: Left-to-right, Leftmost derivation, with *k* symbols look-ahead
- *LR(k)*: Left-to-right, Rightmost derivation, with *k* symbols look-ahead

- Process the input "tokens"* from left to right.

*: *Lexing* - Map, say "temp", to **id**, and 12.3 to **num** etc.
Maps from infinite space to finite space.

- First derive the leftmost variable v. first derive the rightmost symbol

- Top-down (from root to leaf) Parsing v. Bottom-up Parsing (from leave to root)
  - Recall CYK is bottom-up

Look-ahead Parsing – 2

- Why look ahead? Ambiguity and Prediction
  - Productions: $S \rightarrow ab \mid ba$

  - Productions: $S \rightarrow aaP \mid abQ$
  - Input: aa…
  - Which one rule to go to?

  - Productions: id_list → **id** | id_list **, id**
  - Input: temp1, temp2, temp3
  - Which rules** to apply?

**: Need to modify rules to eliminate "left-recursions".

LL(1) Parsing Idea – 1

- Parsing (Syntax)

1. **Pre-process** all viable <production, look-ahead> tuples and construct "Parsing Table".
   - Recursive construction, may take a long time to construct.
   - $\sim O(|P|^4) + O(|V| \cdot |T|)$

   - Only needs to be done **once** per grammar update. JDK 10 → JDK 11
   - Comes with IDE and/or compiler.

   ➢ Requires grammar to be in the class LL(1): one symbol look-ahead w/o ambiguity.

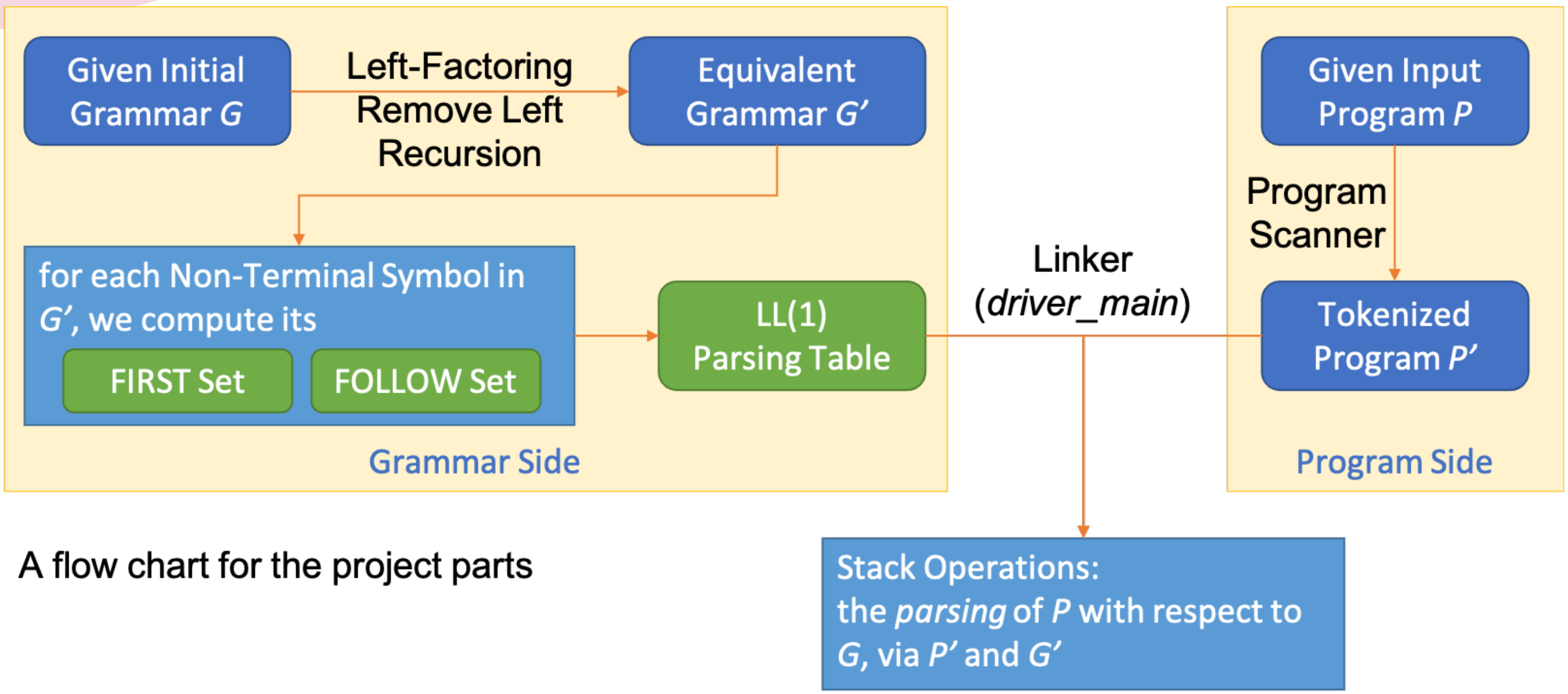
LL(1) Parsing Idea – 2

- Parsing (Syntax)

2. **Parsing**: When parsing a program, **simply look-up** for proper table entries and modify the parsing stack based on “Actions” in the corresponding entries.
    - Only need $O(n)$ time to HALT and tell whether there’s a syntax error or not.
    - Better parsers can tell what the errors are and how to fix them.

LL(1) Parsing Steps

* From Fall 2020



A flow chart for the project parts

LL(1) Parsing Idea – 3

- What's next? (Recall: CSCI 2461 Computer Architecture I)

3. Optimization and resolve Semantics: evaluations before runtime
4. Lastly, translate into machine language ISA

LL(1) Parsing Goal

- Construct a look-up table, "Parser", such that given

<current input token, current top of stack> i.e., one symbol look-ahead

such that, when parsing a string,

we will be able to deterministically decide the proper action,

whether it is to derive rules, match terminals, and accept or decline input.

id_list → **id** | id_list , **id** ⟷ (Equivalent LL(1) Grammar) id_list → **id** id_list'
id_list' → , **id** id_list' | λ

Input: temp1, temp2, temp3

Tokens: **id** , **id** , **id**

TOS: id_list

Lookahead: **id**

LL(1) Parsing Example – 1

R0: $E \rightarrow T\ E'$
R1: $E' \rightarrow +T\ E'$ R2: $E' \rightarrow \lambda$
R3: $T \rightarrow F\ T'$
R4: $T' \rightarrow \times F\ T'$ R5: $T' \rightarrow \lambda$
R6: $F \rightarrow id$ R7: $F \rightarrow (\ E\ )$

Input: id+id×id
Input': id+id×idZ

Parser	id	+	×	(	)	Z
E	R0			R0		
E'		R1			R2	R2
T	R3			R3		
T'		R5	R4		R5	R5
F	R6			R7		

Reject if falls in these entries etc.

Stack	Input	Action
ZE	id+id×idZ	R0
ZE'T	id+id×idZ	R3
ZE'T'F	id+id×idZ	R6
ZE'T'id	id+id×idZ	Match
…	…	…
Z	Z	Accept

*: Need to know the first symbol in E, T, F are **id**; i.e., look ahead on the first symbol of the TOS.

LL(1) Parsing Example – 2

R0: $E \rightarrow T\ E'$
R1: $E' \rightarrow +T\ E'$ R2: $E' \rightarrow \lambda$
R3: $T \rightarrow F\ T'$
R4: $T' \rightarrow \times F\ T'$ R5: $T' \rightarrow \lambda$
R6: $F \rightarrow id$ R7: $F \rightarrow (\ E\ )$

Parser	id	+	×	(	)	Z
E	R0			R0		
E'		R1			R2	R2
T	R3			R3		
T'		R5	R4		R5	R5
F	R6			R7		

Input: id+id×id

Input': id+id×idZ

Stack	Input	Action
ZE	id+id×idZ	R0
ZE'T	id+id×idZ	R3
ZE'T'F	id+id×idZ	R6
ZE'T'id	id+id×idZ	Match
ZE'T'	+id×idZ	R5*
ZE'	+id×idZ	R1
…	…	…
Z	Z	Accept

*: Need to know the first symbol in E' is **+**; i.e., look ahead on the first symbol of the following.

LL(1) Parsing Notable Steps – 1

➢ How are all these related what we've learned so far?? Membership Problem

Program side

- Lexing (Tokenization)
  - Not dealing with "temp", "sum", "ptr_1", etc.; only need to know they are identifiers, **id**.
  - Same with numbers, either int or real.

  - Overall, map everything we read from the input (infinite space, say ASCII*)
  to defined tokens, i.e., set of terminals (finite space).

  - Now, this becomes feasible for hardware implementation. How??

LL(1) Parsing Notable Steps – 2

➢ How are all these related what we've learned so far?? <u>Membership Problem</u>

Program side

- Lexing (Tokenization)
  - How? **Run a DFA**!! [Recall the email address checking application.]
  - Read char by char; return a token whenever a full token is read, until EOF.
  - ➢ for v. for_0 temp_1 v. temp-1 "." in temp.length() v. temp = 1.4
  - How? **Regexes**!! Library: (F)lex

  ➢ Strings are of the form [a-zA-Z][a-zA-Z0-9_]* and are not reserved words are variable names.
    - ✓ A Lexer let us define what a variable name should look like.

LL(1) Parsing Notable Steps – 3

Grammar side

- Convert to LL(1) Grammar
  - Quite similar to our cleaning up procedures: also based on Theorem 6.1

- Construct tables for look-ahead symbols
  - FIRST & FOLLOW Tables for each variable: FIRST(E) = FIRST(T) = FIRST(F) = {**id**, **(**}
- Construct Parser from the two tables
  - Only one production in one table entry (i.e., per ID): finally, **Deterministic PDA**!!
  - Otherwise, called "conflict"; implementation error or Grammar is not LL(1).

- Feed token stream into the parser with a parsing stack
  - Reached an empty entry? **Syntax Error**!!
  - Better parsers tell us how to resolve: mainly syntactically, could also be semantically.

Furthermore

- LL(k) Parsing
  - Can parse grammars in the class of LL(k)
  - From LL(1) to LL(2), further consider the 2nd FIRST and FOLLOW; and inductively build to LL(k).
  - LL(k) parser is contained in LL(k+1) parser

- Power of LL(1) Grammars
  - Already quite powerful: for, if, while, subroutines, recursion, etc.
  - A grammar with ~ 50 rules, 30 variables and 30 terminals: Binary Search, Merge Sort, etc.

- LR(k) Parsing
  - Many modern programming languages uses LR parsing; they do have left-recursions.
  - Even more powerful than the corresponding LL(k) parser.

[Further reading]: Levine, J. "flex & bison," O'Reilly, 2019. Flex & bison following yacc