CS406: Compilers

Spring 2020

Week 5: Parsers, AST, and Semantic Routines

Recap

What is parsing

- Parsing is recognizing members in a language specified/defined/generated by a grammar
- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will take some action
  - In a compiler, this action generates an intermediate representation of the program construct
  - In an interpreter, this action might be to perform the action specified by the construct. Thus, if *a+b* is recognized, the value of *a* and *b* would be added and placed in a temporary variable

Top-down Parsing – predictive parsers

- Idea: we know sentence has to start with initial symbol
- Build up partial derivations by *predicting* what rules are used to expand non-terminals
  - Often called *predictive parsers*
- If partial derivation has terminal characters, *match* them from the input stream

Suggested reading: https://en.wikipedia.org/wiki/LL_parser

Top-down Parsing – contd..

- Also called recursive-descent parsing
- Equivalent to finding the left-derivation for an input string
  - Recall: expand the leftmost non-terminal in a parse tree
  - Expand the parse tree in pre-order i.e. identify parent nodes before children

Top-down Parsing

```
1) S -> F
2) S -> (S + F)
3) F -> a
```

string: `(a+a)`

string': `(a+a)$`

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Assume that the table is given.

- Table-driven (Parse Table) approach doesn't require backtracking

But how do we construct such a table?

First and follow sets

- First(α): the set of terminals (and/or λ) that begin all strings that can be derived from α
  - First(A) = {x, y, λ}
  - First(xaA) = {x}
  - First (AB) = {x, y, b}
- Follow(A): the set of terminals (and/or \$, but no λs) that can appear immediately after A in some partial derivation
  - Follow(A) = {b}

$S \rightarrow A\ B\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow \lambda$

$B \rightarrow b$

First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \ldots A\ \$\}$

S:	start symbol
a:	a terminal symbol
A:	a non-terminal symbol
α,β:	a string composed of terminals and non-terminals (typically, α is the RHS of a production

$\Rightarrow$:	derived in 1 step
$\Rightarrow^*$:	derived in 0 or more steps
$\Rightarrow^+$:	derived in 1 or more steps

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production P (of the form $A \rightarrow X_1X_2 \ldots X_m$) applies

$$\text{Predict}(P) =$$

$$\begin{cases} \text{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\ (\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

- If next token is in Predict(P), then we should choose this production

Computing Parse-Table

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

```
first (S) = {x, y, c}
first (A) = {x, y, c}
first(B) = {b, λ}
```

```
follow (S) = {}
follow (A) = {b, c}
follow(B) = {c}
```

```
P(1) = {x,y,c}
P(2) = {x}
P(3) = {y}
P(4) = {c}
P(5) = {b}
P(6) = {c}
```

Parsing using stack-based model (non-recursive) of a predictive parser

Computing Parse-Table

string: xacc$

Stack*	Remaining Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	match(c)
c$	c$	Done!

* Stack top is on the left-side (first character) of the column

Identifying LL(1) Grammar

- What we saw was an example of LL(1) Parser
- Not all Grammars are LL(1)

A Grammar is LL(1) iff for a production A -> α | β, where α and β are distinct:

1. For no terminal a do both α and β derive strings beginning with a
2. At most one of α and β can derive an empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in `Follow(A)`. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then does not derive any string beginning with a terminal in `Follow(A)`

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  $E \rightarrow E + T$

- What would happen with the stack-based algorithm?

Example (Left Factoring)

- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif

- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Eliminating Left Recursion

$$A \text{ -> } A\ \alpha \mid \beta$$

$$\Downarrow$$

$$A \text{ -> } \beta A'$$
$$A' \text{ -> } \alpha A' \mid \lambda$$

LL(k) parsers

- Can look ahead more than one symbol at a time
  - *k*-symbol lookahead requires extending first and follow sets
  - 2-symbol lookahead can distinguish between more rules:

    $A \rightarrow ax \mid ay$
- More lookahead leads to more powerful parsers
- What are the downsides?

Are all grammars LL(k)?

- No! Consider the following grammar:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow (E + E) \\ E &\rightarrow (E - E) \\ E &\rightarrow x \end{aligned}$$

- When parsing E, how do we know whether to use rule 2 or 3?
  - Potentially unbounded number of characters before the distinguishing '+' or '–' is found
  - No amount of lookahead will help!

In real languages?

- Consider the if-then-else problem
- `if x then y else z`
- Problem: else is optional
- `if a then if b then c else d`
    - Which `if` does the `else` belong to?
- This is analogous to a "bracket language": $[^i\ ]^j\ (i \geq j)$

$$
\begin{aligned}
S &\rightarrow [\ S\ C \\
S &\rightarrow \lambda \\
C &\rightarrow\ ] \\
C &\rightarrow \lambda
\end{aligned}
$$

[[ ] can be parsed: SSλC or SSCλ
(it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
  - "] matches nearest unmatched ["
  - This is the rule C uses for if-then-else
  - What if we try this?

$$
\begin{aligned}
S &\rightarrow [\ S \\
S &\rightarrow S1 \\
S1 &\rightarrow [\ S1\ ] \\
S1 &\rightarrow \lambda
\end{aligned}
$$

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match “]” before matching “λ”

$$
\begin{aligned}
S &\rightarrow [\ S\ C \\
S &\rightarrow \lambda \\
C &\rightarrow] \\
C &\rightarrow \lambda
\end{aligned}
$$

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

$$
\begin{aligned}
S &\rightarrow \text{if } S\ E \\
S &\rightarrow \text{other} \\
E &\rightarrow \text{else } S \text{ endif} \\
E &\rightarrow \text{endif}
\end{aligned}
$$

Parsing if-then-else

- What if we don't want to change the language?
  - C does not require { } to delimit single-statement blocks
- To parse if-then-else, *we need to be able to look ahead at the entire rhs of a production* before deciding which production to use
  - In other words, we need to determine how many "]" to match before we start matching "["s
- *LR parsers* can do this!

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
  - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves

Example:

```
E -> E + T | T
T -> T * F | F
F -> (E) | id
```

String: `id*id`

Demo

LR Parsers

• Basic idea: put tokens on a stack until an entire production is found

- **shift** tokens onto the stack. At any step, keep the set of productions that could generate the read-in token
- **reduce** the RHS of recognized productions to the corresponding non-terminal on the LHS of the production. Replace the RHS tokens on the stack with the LHS non-terminal.

• Issues:

- Recognizing the endpoint of a production
- Finding the length of a production (RHS)
- Finding the corresponding nonterminal (the LHS of the production)

Data structures

- At each state, given the next token,
  - A *goto table* defines the successor state
  - An *action table* defines whether to
    - *shift* – put the next state and token on the stack
    - *reduce* – an RHS is found; process the production
    - *terminate* – parsing is complete

Simple example

1. P → S
2. S → x ; S
3. S → e

		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Parsing using an LR(0) parser

- Basic idea: parser keeps track, simultaneously, of all possible productions that *could be matched* given what it's seen so far. When it sees a full production, match it.
- Maintain a *parse stack* that tells you what state you're in
  - Start in state 0
- In each state, look up in action table whether to:
  - *shift*: consume a token off the input; look for next state in goto table; push next state onto stack
  - *reduce*: match a production; pop off as many symbols from state stack as seen in production; look up where to go according to non-terminal we just matched; push next state onto stack
  - *accept*: terminate parse

Example

- Parse "x ; x ; e"

Step	Parse Stack	Remaining Input	Parser Action
1	0	x ; x ; e	Shift 1
2	0 1	; x ; e	Shift 2
3	0 1 2	x ; e	Shift 1
4	0 1 2 1	; e	Shift 2
5	0 1 2 1 2	e	Shift 3
6	0 1 2 1 2 3		Reduce 3 (goto 4)
7	0 1 2 1 2 4		Reduce 2 (goto 4)
8	0 1 2 4		Reduce 2 (goto 5)
9	0 5		Accept

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in *pre-order*
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in *post-order*
  - Identify children before the parents
- Notation:
  - LL(1): Top-down derivation with 1 symbol lookahead
  - LL(k): Top-down derivation with k symbols lookahead
  - LR(1): Bottom-up derivation with 1 symbol lookahead

Abstract Syntax Trees

- Parsing recognizes a production from the grammar based on a sequence of tokens received from Lexer
- Rest of the compiler needs more info: a structural representation of the program construct
  - Abstract Syntax Tree or AST

Abstract Syntax Trees

- Are like parse trees but ignore certain details
- Example:

E -> E + E | (E) | int

String: 1 + (2 + 3)

Demo

Semantic Actions for Expressions

Review

- Scanners
  - Detect the presence of illegal tokens
- Parsers
  - Detect an ill-formed program
- Semantic actions
  - Last phase in the *front-end* of a compiler
  - Detect all other errors

What are these kind of errors?

What we cannot express using CFGs

- Examples:
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - etc.

  Depends on the language..

Semantic actions

- *Semantic actions* are routines called as productions (or parts of productions) are recognized
- Actions work together to build up intermediate representations
- Conceptually think of this as follows:
  - Every non-terminal should have some information associated with it (code, declared variables, etc.)
  - Each child of a non-terminal can pass the information it has to its parent non-terminal, which uses the information from its children to build up more information
  - We call these *semantic records*

Semantic Records

- Data structures produced by semantic actions
- Associated with both non-terminals (code structures) and terminals (tokens/symbols)
- Build up semantic records by performing a bottom-up walk of the abstract syntax tree

Scope

- *Scope* of an identifier is the part of the program where the identifier is accessible
- Multiple scopes for same identifier name possible
- Static vs. Dynamic scope

exercise: what are the different scopes in Micro?

Types

- Static vs. Dynamic
- Type checking
- Type inference

Referencing identifiers

- What do we return when we see an identifier?
  - Check if it is in symbol table
  - Create new AST node with pointer to symbol table entry
  - Note: may want to directly store type information in AST (or could look up in symbol table each time)

Referencing Literals

- What about if we see a literal?

  primary → INTLITERAL | FLOATLITERAL

- Create AST node for literal
- Store string representation of literal
  - "155","2.45" etc.
- At some point, this will be converted into actual representation of literal
  - For integers, may want to convert early (to do *constant folding*)
  - For floats, may want to wait (for compilation to different machines). Why?

Expressions

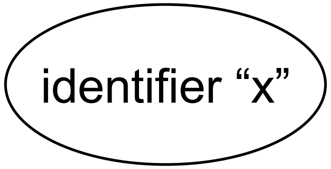
- Three semantic actions needed
  - eval_binary (processes binary expressions)
    - Create AST node with two children, point to AST nodes created for left and right sides
  - eval_unary (processes unary expressions)
    - Create AST node with one child
  - process_op (determines type of operation)
    - Store operator in AST node

Expressions Example

```
x + y + 5
```

Expressions Example

x + y + 5

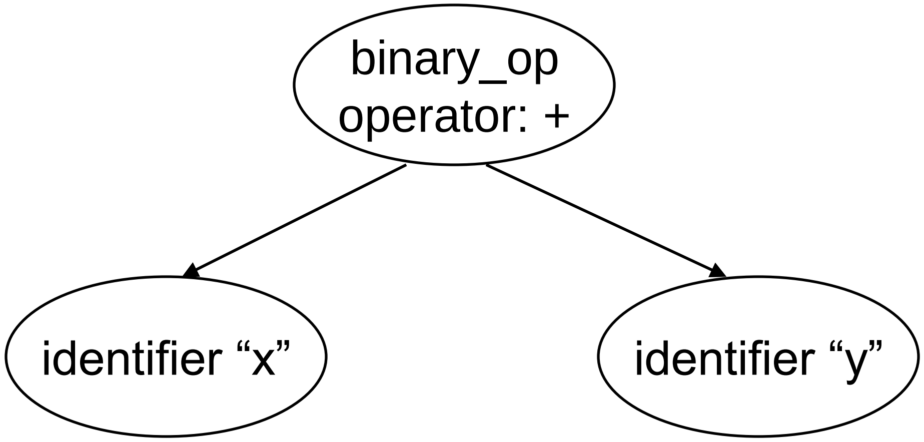


Expressions Example

x + y + 5

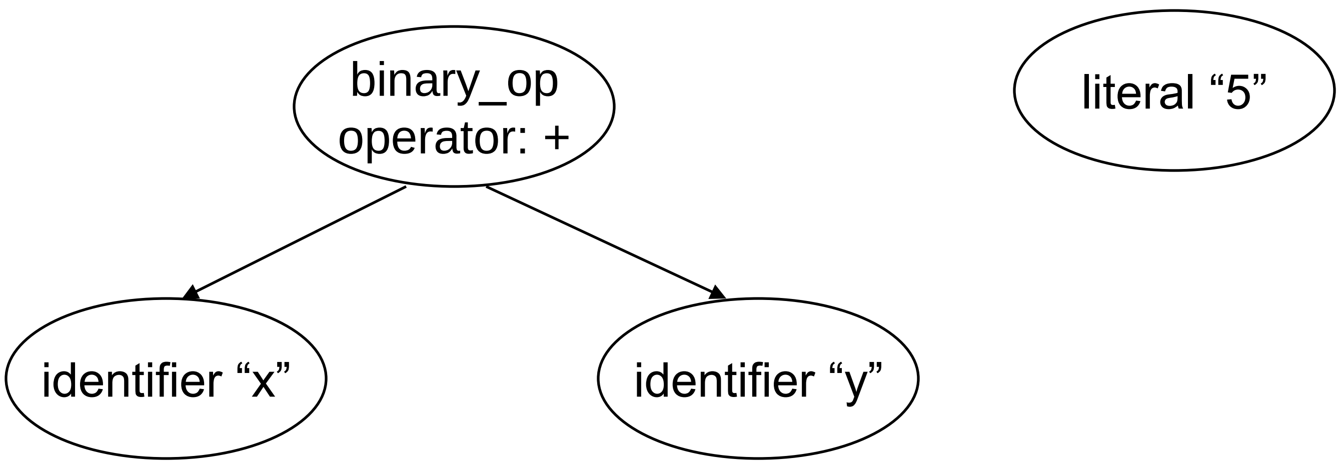
Expressions Example

`x + y + 5`



Expressions Example

```
x + y + 5
```



Expressions Example

```
x + y + 5
```