Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Top-Down Parsing

Top-down Parsing Algorithm

Construct the root node of the parse tree, label it with the start symbol, and set the current-node to root node

Repeat until all the input is consumed (i.e., until the frontier of the parse tree matches the input string)

1. If the label of the current node is a non-terminal node A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
2. If the current node is a terminal symbol:
   - If it matches the input string, consume it (advance the input pointer)
   - If it does not match the input string, backtrack
3. Set the current node to the next node in the frontier of the parse tree
   - If there is no node left in the frontier of the parse tree and input is not consumed, then backtrack

The key is picking the right production in step 1

- That choice should be guided by the input string

Predictive Parsing

- The main idea is to look ahead at the next token and use that token to pick the production that you should apply

$$X \rightarrow + X \;|\; - Y$$

Here we can use the + and – to decide which rule to apply

This technique is more general!

- Definition of FIRST sets

$x \in \mathrm{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* x\,\gamma$, for some γ

($\Rightarrow^*$ means a series of (0 or more) productions)

- This means that we have to examine ALL tokens that our productions could potentially start with

FIRST Sets

- Intuitively, FIRST(S) is the set of all terminals that we could possibly see when starting to parse S
- If we want to build a predictive parser, we need to make sure that the look-ahead token tells us with 100% confidence which production to apply
- In order for this to be true, anytime we have a production that looks like $A \rightarrow \alpha \mid \beta$, we need to make sure that FIRST(α) is distinct from the FIRST(β)
- "Distinct" means that there is no element in FIRST(α) that is also in FIRST(β) … or formally, that FIRST(α) $\cap$ FIRST(β) = {}

Slightly More Tricky Examples

S → AB
A → x | y
B → 0 | 1

FIRST(S) = { x, y }

S → AB
A → x | y | ε
B → 0 | 1

FIRST(S) = { x, y, 0, 1 }

S → AB
A → x | y | ε
B → 0 | 1 | ε

FIRST(S) = { x, y, 0, 1, ε }

- Here is an example of FIRST sets where the first symbol in the production is a non-terminal
- In this case, we have to examine *all* possible terminals that could begin a sentence derived from S
- If we have an ε, then we need to look past the first non-terminal
- If all the non-terminals have ε in their first sets, then add ε to the first set

How to Generate FIRST Sets

For a string of grammar symbols α, define FIRST(α) as

- Set of tokens that appear as the first symbol in some string that derives from α
- If $\alpha \Rightarrow^* \varepsilon$, then ε is in FIRST(α)

To construct FIRST(X) for a grammar symbol X, apply the following rules until no more symbols can be added to FIRST(X)

- If X is a terminal, then FIRST(X) is $\{X\}$
- If $X \rightarrow \varepsilon$ is a production, then ε is in FIRST(X)
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put every symbol in FIRST(Y_1) other than ε to FIRST(X)
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put terminal a in FIRST(X) if a is in FIRST(Y_i) and ε is in FIRST(Y_j) for all $1 \leq j < i$
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put ε in FIRST(X) if ε is in FIRST(Y_i) for all $1 \leq i \leq k$

Computing FIRST Sets

To construct the FIRST set for any string of grammar symbols $X_1X_2 \ldots X_n$ (given the FIRST sets for symbols X_1 , X_2 , ... X_n), apply the following rules.

FIRST($X_1X_2 \ldots X_n$) contains:

- any symbol in FIRST(X_1) other than ε
- any symbol in FIRST(X_i) other than ε, if ε is in FIRST(X_j) for all $1 \leq j < i$
- ε, if ε is in FIRST(X_j) for all $1 \leq i \leq n$

FIRST Sets

1	*S*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	num
11		|	id

Symbol	FIRST
S	{num, id}
Expr	{num, id}
Expr′	{ε , +, - }
Term	{num, id}
Term′	{ε , *, / }
Factor	{num, id}

We still have those pesky epsilons …

$S \rightarrow AB$

$A \rightarrow x \mid y \mid \varepsilon$

$B \rightarrow 0 \mid 1 \mid \varepsilon$

FIRST(S) = { x, y, 0, 1, ε }

- Despite our efforts to look past all of the ε when defining our FIRST sets, sometimes we still have ε in our FIRST sets (as in the above example). So, what can we do?
- The trick to doing it is to look past the current non-terminal and examine the set of characters that can *follow* the current non-terminal
- This is what the FOLLOW set defines
- We use the special character $ to denote the end of the file

FOLLOW Sets

For a non-terminal symbol *A*, define FOLLOW(*A*):

The set of terminal symbols that can appear immediately to the right of *A* in some sentential form

To construct FOLLOW(*A*) for a non-terminal symbol *A*, apply the following rules until no more symbols can be added to FOLLOW(*A*):

- Place $ in FOLLOW(*S*) ($ is the end-of-file symbol, *S* is the start symbol)
- If there is a production $A \rightarrow \alpha\ B\ \beta$, then everything in FIRST(β) - except ε - is placed in FOLLOW(*B*)
- If there is a production $A \rightarrow \alpha\ B$, then everything in FOLLOW(*A*) is placed in FOLLOW(*B*)
- If there is a production $A \rightarrow \alpha\ B\ \beta$, and ε is in FIRST(β), then everything in FOLLOW(*A*) is placed in FOLLOW(*B*)

FOLLOW Sets

1	*S*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	num
11		|	id

Symbol	**FOLLOW**
S	{ $ }
Expr	{ $ }
Expr′	{ $ }
Term	{ $, +, - }
Term′	{ $, +, - }
Factor	{ $, +, -, *, / }

Another FIRST/FOLLOW Example

Example Input: x + y (z + a (b))

```
Expression  →  Function
            |   ( Expression )
            |   Primary + Expression
            |   Primary
Primary     →  id
            |   integer
Function    →  id ( ParamList )
ParamList   →  Expression ParamList
            |   ε
```

FIRST (Expression) = { **(**, **integer, id** }
FIRST (Primary) = { **integer, id** }
FIRST (Function) = { **id** }
FIRST (ParamList) = { **(**, **id**, **integer,** ε }

FOLLOW (Expression) = { **$** , **(** , **)** , **id, integer** }
FOLLOW (Primary) = { **$** , **(** , **)** , **+** , **id, integer** }
FOLLOW (Function) = { **$** , **(** , **)** , **id, integer** }
FOLLOW (ParamList) = { **)** }

LL(1) Grammars

Left-to-right scan of the input, **L**eftmost derivation, **1**-token look-ahead

A grammar *G* is LL(1) if for each set of its productions
$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

FIRST(α_1), FIRST(α_2), ..., FIRST(α_n), are all pair-wise disjoint

If $\alpha_i \Rightarrow^* \varepsilon$, then FIRST (α_j) $\cap$ FOLLOW (A) = $\varnothing$ for all $1 \leq i \leq n$, $i \neq j$

- In other words, LL(1) grammars
  - productions are uniquely predictable given a context (look-ahead)
  - cannot have left recursion (direct or indirect)

Recursive Descent Parsing

- Use a set of *mutually recursive* procedures
  - one procedure for each non-terminal symbol
  - start the parsing process by calling the procedure that corresponds to the start symbol
  - each production becomes one clause in procedure
- Use a look-ahead symbol to decide which production to use
  - based on the elements in the FIRST sets
- When no element in FIRST set matches, check the FOLLOW set
  - if look-ahead symbol is in FOLLOW set and there is an epsilon production, return from procedure (i.e., take epsilon production)
  - otherwise, terminate with a parsing error

Recursive Descent Parsing

1	S	$\rightarrow$	if E then S else S
2		|	begin S L
3		|	print E
4	L	$\rightarrow$	end
5		|	; S L
6	E	$\rightarrow$	num = num

```
void match(int token) {
  if (lookahead==token)
    lookahead=getNextToken();
  else
    error();
}

void main() {
  lookahead=getNextToken();
  S();
  match(EOF);
}
```

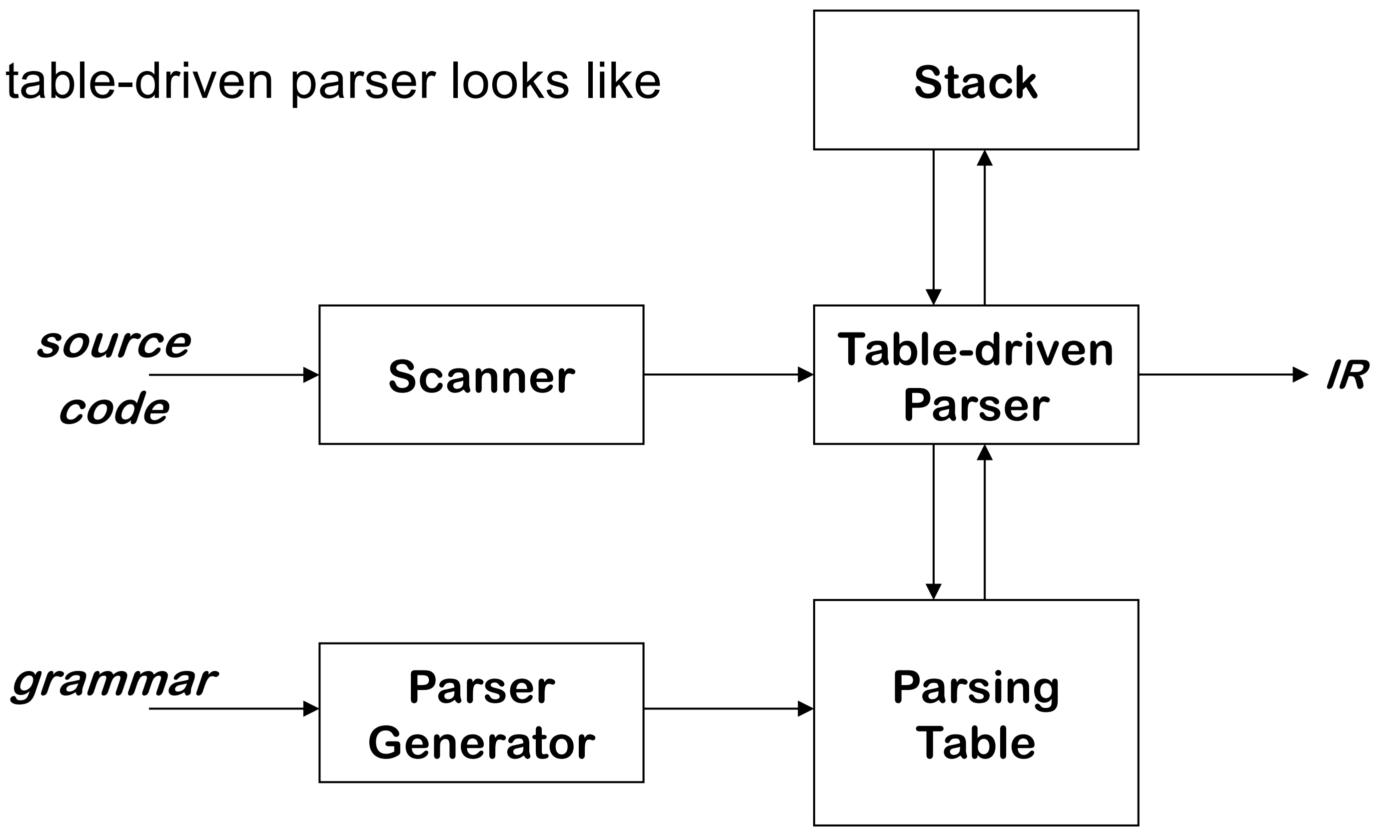
```
void S() {
  switch(lookahead) {
    case IF: match(IF); E(); match(THEN); S();
             match(ELSE); S(); break;
    case BEGIN: match(BEGIN); S(); L(); break;
    case PRINT: match(PRINT); E(); break;
    default: error();
  }
}

void L() {
  switch(lookahead) {
    case END: match(END); break;
    case SEMI: match(SEMI); S();
               L(); break;
    default: error();
  }
}

void E() { match(NUM); match(EQ); match(NUM); }
```

Alternative: Table-Driven Parsers

A table-driven parser looks like



Parsing tables can be built automatically!

Stack-Based, Table-Driven Parsing

The parsing table

- A two-dimensional array

  M[*A*, *a*] $\rightarrow$ gives a production

  A: non-terminal symbol

  a: terminal symbol

- What does it mean?
  - If top of the stack is *A* and the look-ahead symbol is *a,* then we apply the production M[*A, a*]

	IF	BEGIN	PRINT	END	SEMI	NUM
S	$S\rightarrow$if E then S else S	$S\rightarrow$begin S L	$S\rightarrow$print E			
L				$L\rightarrow$ end	$L\rightarrow$; S L	
E						$E\rightarrow$num = num

Table-Driven Parsing Algorithm

- Push the end-of-file symbol (**$**) and the start symbol *S* onto the stack
- Consider the symbol *X* on the top of the stack and look-ahead (terminal) symbol *a*
  - If *X* = $ and a = $, then announce successful parse and halt
  - If *X* = ***a*** (and **a** ≠ $), pop *X* off the stack and advance the input pointer to the next input symbol (read in new **a**)
  - If *X* is a non-terminal, look at the production M[*X*, ***a***]
    - If there is no such production (M[*X*, *a*] = error), then call an error routine
    - If M[*X*, *a*] is a production $X \rightarrow Y_1\ Y_2 \ldots Y_k$, then pop *X* and push Y_k , Y_{k-1} , ..., Y_1 onto the stack with Y_1 on top
  - If none of the cases above apply, then call an error routine

Table-Driven Parsing Algorithm


```
Push($);   // $ is the end-of-file symbol
Push(S);   // S is the start symbol of the grammar
lookahead = get_next_token();
repeat
    X = top_of_stack();
    if (X is a terminal or X == $) then
        if (X == lookahead) then
            pop(X);
            lookahead = get_next_token();
        else error();
    else  // X is a non-terminal
         if ( M[X, lookahead] == X → Y1 Y2 ... Yk) then
             pop(X);
             push(Yk); push(Yk-1); ... push(Y1);
         else error();
until (X = $)
```

Table-Driven : if 2=2 then print 5=5 else print 1=1$

Stack	**lookahead**	**Parse-table lookup**
$,*S*	IF	M[*S*,IF]: *S*→if *E* then *S* else *S*
$,*S*,ELSE,*S*,THEN,*E*,IF	IF	
$*S*,ELSE,*S*,THEN,*E*	NUM	M[*E*,NUM]: *E*→num = num
$*S*,ELSE,*S*,THEN,NUM,EQ,NUM	NUM	
$*S*,ELSE,*S*,THEN,NUM,EQ	EQ	
$*S*,ELSE,*S*,THEN,NUM	NUM	
$*S*,ELSE,*S*,THEN	THEN	
$*S*,ELSE,*S*	PRINT	M[*S*,PRINT]: *S*→print *E*
$*S*,ELSE,*E*,PRINT	PRINT	
$*S*,ELSE,*E*	NUM	M[*E*,NUM]: *E*→num = num
$*S*,ELSE,NUM,EQ,NUM	NUM	
$*S*,ELSE,NUM,EQ	EQ	
$*S*,ELSE,NUM	NUM	
$*S*,ELSE	ELSE	
$*S*	PRINT	M[*S*,PRINT]: *S*→print *E*
$*E*,PRINT	PRINT	
$*E*	NUM	M[*E*,NUM]: *E*→num = num
$NUM,EQ,NUM	NUM	
$NUM,EQ	EQ	
$NUM	NUM	
$	$	report success!

LL(1) Parse Table Construction

- For all productions $A \rightarrow \alpha$, perform the following steps:
  - For each terminal symbol a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a]
  - If ε is in FIRST(α), then add $A \rightarrow \alpha$ to M[A, b] for each terminal symbol b in FOLLOW(A).
  - Add $A \rightarrow \alpha$ to M[A, \$] if \$ is in FOLLOW(A)
- Set all the undefined entries in M to ERROR

LL(1) Parse Table Construction

Grammar:

1	***S***	→	***Expr***
2	***Expr***	→	***Term Expr′***
3	***Expr′***	→	**+ *Term Expr′***
4		|	**- *Term Expr′***
5		|	ε
6	***Term***	→	***Factor Term′***
7	***Term′***	→	*** *Factor Term′***
8		|	**/ *Factor Term′***
9		|	ε
10	***Factor***	→	**num**
11		|	**id**

Symbol	FOLLOW
S	{ $ }
Expr	{ $ }
Expr′	{ $ }
Term	{ $, +, - }
Term′	{ $, +, - }
Factor	{ $, +, -, *, / }

Symbol	FIRST
S	{num, id}
Expr	{num, id}
Expr′	{ε , +, - }
Term	{num, id}
Term′	{ε , *, / }
Factor	{num, id}
num	{num}
id	{id}
+	{+}
-	{-}
*	{*}
/	{/}

LL(1) Parse Table Construction

LL(1) Parse table:

	id	num	+	-	*	/	$
S	$S \to E$	$S \to E$					
E	$E \to T\,E'$	$E \to T\,E'$					
E'			$E' \to +\,T\,E'$	$E' \to -\,T\,E'$			$E' \to \varepsilon$
T	$T \to F\,T'$	$T \to F\,T'$					
T'			$\boldsymbol{T'} \to \varepsilon$	$\boldsymbol{T'} \to \varepsilon$	$T' \to *\,F\,T'$	$T' \to /\,F\,T'$	$\boldsymbol{T'} \to \varepsilon$
F	$F \to$ id	$F \to$ num					

LL(1) Grammar

Left-to-right scan of the input, **L**eftmost derivation, **1**-token look-ahead

Two alternative definitions of LL(1) grammars:

1. A grammar *G* is LL(1) if there are no multiple entries in its LL(1) parse table

2. A grammar *G* is LL(1), if for each set of its productions
   $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

   FIRST(α_1), FIRST(α_2), ..., FIRST(α_n) are all pair-wise disjoint

   If $\alpha_i \Rightarrow^* \varepsilon$, then FIRST (α_j) $\cap$ FOLLOW (A) = $\emptyset$ for all $1 \leq i \leq n$, $i \neq j$

The Verdict on Top-Down Parsing

- Top down parsers are great
  - They are (relatively) simple to construct by hand
  - They have many real-world applications
  - They provide the most intuitive way to reason about parsing
  - Predictive parsing is fast
- Top down has some problems
  - It can get messy for complex grammars (like full Java)
  - It does not handle left-recursion, which is how we would like to specify left-associative operators
  - It is quite restrictive on the the types of grammars we can parse
- What we need is a fast and automated approach that can handle a more general set of grammars
  - This requires a different way of thinking about parsing ...

Where are we in the process?

UC Santa Barbara