Lecture 15: Lexing & Parsing

CSCI 131
Fall, 2010

Kim Bruce

Final Grammar

```
      <exp> ::= <term> <termTail>                 (1)
  <termTail> ::= <addop> <term> <termTail>        (2)
               | ε                                (3)
     <term> ::= <factor> <factorTail>             (4)
<factorTail> ::= <mulop> <factor> <factorTail>    (5)
               | ε                                (6)
    <factor> ::= ( <exp> )                        (7)
               | NUM                              (8)
               | ID                               (9)
     <addop> ::= + | -                           (10)
     <mulop> ::= * | /                           (11)
```

First & Follow

- *Intuition*: b ∈ First(X) iff there is a derivation $X \rightarrow^* b\omega$ for some ω.
- *Intuition:* A terminal b ∈ Follow(X) iff there is a derivation $S \rightarrow^* vXb\omega$ for some v and ω.

First for Arithmetic

FIRST(<exp>) = { (, NUM, ID }
FIRST(<termTail>) = { +, -, ε }
FIRST(<term>) = { (, NUM, ID }
FIRST(<factorTail>) = { *, /, ε }
FIRST(<factor>) = { (, NUM, ID }
FIRST(<addop>) = { +, - }
FIRST(<mulop>) = { *, / }

Follow for Arithmetic

FOLLOW(<exp>) = { EOF,) }
FOLLOW(<termTail>) = FOLLOW(<exp>) = { EOF,) }
FOLLOW(<term>) = FIRST(<termTail>) ∪ FOLLOW(<exp>) ∪ FOLLOW(<termTail>) = { +, -, EOF,) }
FOLLOW(<factorTail>) = { +, -, EOF,) }
*FOLLOW(<factor>) = { *, /, +, -, EOF }*
FOLLOW(<addop>) = { (, NUM, ID }
FOLLOW(<mulop>) = { (, NUM, ID }
} *Not needed!*

Predictive Parsing, redux

Goal: $a_1a_2...a_n$

$S \rightarrow \alpha$
...
$\rightarrow a_1a_2X\beta$

Want next terminal character derived to be a_3

Need to apply a production X ::= γ where *in First(X)*
1) γ can eventually derive a string starting with a_3 or
2) If X can derive the empty string, then se *in Follow(X)*
if β can derive a string starting with a_3.

Parse Table for Arithmetic

Non-terminals	*ID*	*NUM*	*Addop*	*Mulop*	*(*	*)*	*EOF*
<exp>	1	1			1		
<termTail>			2			3	3
<term>	4	4			4		
<factTail>			6	5		6	6
<factor>	9	8			7		
<addop>			10				
<mulop>				11			

See Haskell Recursive Descent Parser, ParseArith.hs

Table-Driven Stack-based Parser

- http://en.wikipedia.org/wiki/LL_parser
- Start with S $ on stack and input $ to be recognized.
- Use table to replace non-terminals on top of stack.
- If terminal on top of stack matches next input then erase both and proceed.
- Success if end up clearing stack and input
- Show with ID * (NUM + NUM)$

Another alternative

- LR(1) parsers -- bottom up, gives right-most derivation.
- YACC is LR(1). ANTLR is LL(1).
- k in LL(k) and LR(k) indicates how many letters of look ahead are necessary -- e.g. length of strings in columns of table.
- Compiler writers are happiest with k=1 to avoid exponential blow-up of table. May have to rewrite grammars.

Scope -- a bit out of order ...

Scope

- Range of instructions where identifier is known
- Static: Scope associated with static text of program.
- Dynamic: Scope associated with execution path of program.

Hole in Scope (Static)

```
program ...
    var M: integer;
    ....
    procedure A ...
        var M: array [1..10] of real;
        begin
            ...
        end;
begin
    ...
end.
```

Static vs Dynamic Scope

```
program ...
    var A : integer;

    procedure Y(B: integer);
        begin
            ...;
            B := A + B;
            ...
        end; {Y}
```

which A?

```
procedure Z(...);
     var A: integer;
     begin
          ...;
          Y(...);
          ...
     end; {Z}
begin {main}
     ...;
     Z(...);
     ...
end.
```

Symbol Table: Built compile-time or run-time?

Semantics

- Operational
- Axiomatic
- Denotational (*Mitchell text.*)

Natural (*Operational*) Semantics

- Arithmetic expressions example on web page
  - ArithSemantics.hs
- How to interpret identifiers?
- Environment: Association list of id's & values.
- Semantics defined recursively on abstract syntax trees.