Homeworks now due on Wednesdays

Will introduce on Thursdays, discuss on Tuesdays if warranted

Assignment 3: Recognizer for Lake expressions; silently accepts

Written problem: recursive descent or table-driven LL(1) parser OK

Assignment 3 details: defining precedence and associativity

Can be implemented grammatically
Stylistically preferable

Can be implemented with precedence rules
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;

DO NOT use both
Bad style: confusing for future users (yourself)

Rules of thumb for grammatically defining precedence and associativity:

1. Associativity follows recursion
2. Precedence accomplished via "chaining"
Example:

\[ e ::= n \mid e + e \quad \text{(expressions)} \]

We want + to be left-associative:

\[
\begin{align*}
\text{expr} & ::= \text{expr} \text{ PLUS} \text{ constant}; \\
\text{expr} & ::= \text{constant}; \\
\text{constant} & ::= \text{INT\_CONST};
\end{align*}
\]

1 + 2 + 3 is implicitly \((1 + 2) + 3\):

Example:

\[ e ::= n \mid e + e \mid e \ast e \quad \text{(expressions)} \]

Associativity:

*, + left-associative

Precedence:

* has higher precedence than +
Intuition: in chained grammar, lower in the chain means lower in syntax trees

The lower things appear in syntax trees, the higher their precedence

Syntax trees evaluated from the leaves up

Constants have highest precedence, evaluated immediately

```
expr ::= add_expr;
add_expr ::= add_expr PLUS mult_expr;
add_expr ::= mult_expr;
mult_expr ::= mult_expr STAR constant;
mult_expr ::= constant;
constant ::= INT_CONST;
```

1 + 2 + 3 * 4 is implicitly (1 + 2) + (3 * 4):
Suggestions for Assignment 3:

Combine operators into precedence equivalence classes

Define non-terminal for each class

Use chaining and left- or right-recursion to specify precedence and associativity

Next topic: Symbol Tables

Named program entities:
- functions
- variables and parameters
- enum elements
- structs, unions, enum types
- fields
- types (not in lake)
- operators (in C++, others)

Compilers track named entities
Maintain info in a symbol

Symbol info varies by kind of entity

Symbol info may vary by phase
Symbol info can include:

- name
- type
- defining source position
- storage allocated
- initial value
- visibility
- lifetime
- scope
- special usage flags

Each language may define distinct special usage flags

C (and Lake) have only one
- cannot put variable in register

Flagged for unary &
- If address is taken, cannot put variable in register

Scope is portion of program where name is visible

Most common scope in C is block
- Variables declared in block
- Only visible to stmts in block

Other scopes:
- file, function, struct/union
Often multiple valid scopes at once
Only one (current scope) allows definition
Nested scopes

Typical nesting:
file
function
block
All defines go into last block (current scope)

structs and unions define a scope
Only accessible after . or ->
Temporarily displaces all other scopes
“disappears” again after one ID

Symbol table is key data structure
Holds symbols
Must be fast (interactions effect lang. speed)

Sometimes called environment

For each scope:
Compiler defines one (or more) symbol tables
name space is set of names (no repetitions in names)

Scope needs multiple symbol tables only if multiple name spaces

C has 3 name spaces:
  Variables
  Types
  goto labels
Types only at file scope
Labels only at function scope

Symbol tables have standard interface
insert: insert a new symbol
lookup: find definition for symbol
  Must be fast
extend: create a new nested scope
retract: dispose extended scope

Start with empty file scope

```c
int g = 3;

void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
}
```
Insert variable into scope

```c
int g = 3;

void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0 / g;
        ...
    }
}
```

Create symbol table for function scope

```c
int g = 3;

void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0 / g;
        ...
    }
    // new s.t. sees old symbols
}
```

Insert parameter into scope

```c
int g = 3;

void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0 / g;
        ...
    }
    // 2 names now visible
}
```
Extend scope for body block

```c
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
}
```

Insert variable b

```c
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
}
```

Extend for new block

```c
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        extend
        float a = 1.0/g;
        ...
    }
}
```
Insert variable a

```
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
    new a masks old a
}
```

Finish block

```
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
    retract
    old a reappears
}
```

Finish function body

```
int g = 3;
void f(int a)
{
    int b = a == 0;
    while (b)
    {
        float a = 1.0/g;
        ...
    }
    retract, retract
    only g visible now
```
Need to implement symbol tables.

*Linked-lists* provide good naïve implementation.

Extend with push, retract with pop

But: lookup too *slow* (linear in the size of the number of symbols)

Best implementation is hash table

Provides *fast* lookup and insertion (constant time)

But how to extend/retract?

Consider single hash table first
Insert adds entry at start of chain
Overflow points to previous entry

extend copies spine

insert still inserts at beginning
retract goes back to old spine
“loses” new definition

Hash table needs hashing function
Converting key into integer

Goal is even distribution (otherwise performance same as linked-list)

Hard to find good, well distributed hashing function for programs
Always diabolical cases
Programmers seem to want to find them