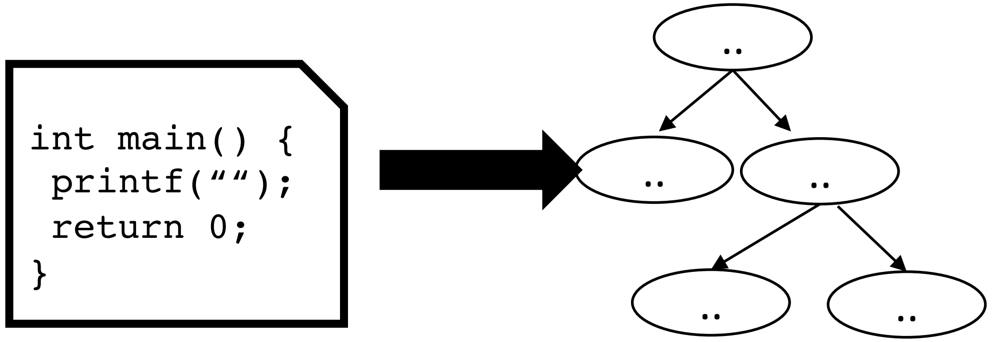
CSE110A: Compilers

April 22, 2022

Topics:

- *Symbol Tables in parsing*
- *Parsing actions*
- *Parser generators*



Announcements

- HW 2 is out!
  - due on May 2 at midnight
  - You had everything for part 1 and 2 after wednesday
  - You will have everything you need for part 3 after today
  - Plenty of chances for help. Get started early

- Midterm will be given on May 2
  - Take home midterm.
  - Assigned on Monday and due on Friday
  - No late midterms are accepted

- No class on Monday (use the time to work on homework)

Announcements

- Expect HW 1 grades around May 2
  - You have 2 weeks to do the homework and we get 2 weeks to grade it

Announcements

- HW 2 clarifications:
  - No skeleton for part 1 - it is done completely in your report
  - Please read the piazza for questions about the grammar and other hints

– An assignment statement, which is ID followed by = followed by an expression.

An assignment statement is followed by a semi colon. The language is a subset of C. Anything that C-simple accepts should also be accepted by C (with the same meaning).

Announcements

- Some more homework examples:
  - Variable declarations vs. assignment statements
  - for statements
  - block statements

Quiz

Is the following grammar backtrack free?

A → B a

B → d a b

| C b

C → c B

| A c

Quiz

First sets

A → B a	{}
B → d a b	{}
| C b	{}
C → c B	{}
| A c	{}

Quiz

First sets

A → B a	{d,c}
B → d a b	{d}
| C b	{d,c}
C → c B	{c}
| A c	{d,c}

no! in both B and C we do not have disjoint first sets

Quiz

Is the following grammar backtrack free?

$A \rightarrow B\ a$

$B \rightarrow d\ a\ b$

$\quad |\ C\ b$

$C \rightarrow c\ B$

$\quad |\ D$

$D \rightarrow d\ B$

Quiz

First sets

$A \rightarrow B\ a$	{}
$B \rightarrow d\ a\ b$	{}
$\mid C\ b$	{}
$C \rightarrow c\ B$	{}
$\mid D$	{}
$D \rightarrow d\ B$	{}

Quiz

	First sets
A → B a	{c,d}
B → d a b	{d}
| C b	{c,d}
C → c B	{c}
| D	{d}
D → d B	{d}

No, because for production B the first sets are not disjoint

Quiz

in a recursive descent parser, you make a function for each or what?

- ○ production option
- ○ CFG
- ○ non-terminal
- ○ terminal

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr?

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr?
We parse a Unit followed by an Expr2

We can just write exactly that!

```
def parse_Expr(self):
        self.parse_Unit();
        self.parse_Expr2();
        return
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr2?

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr2?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op     ::= ‘+’
7:          |  ‘*’
```

How do we parse an Expr2?

```
def parse_Expr2(self):

    token_id = get_token_id(self.to_match)

    # Expr2 ::= Op Unit Expr2
    if token_id in ["PLUS", "MULT"]:
        self.parse_Op()
        self.parse_Unit()
        self.parse_Expr2()
        return

        # Expr2 ::= ""
    if token_id in [None, ”RPAR"]:
        return

    raise ParserException(-1,                            # line number (for you to do)
                          self.to_match,                 # observed token
                          ["PLUS", "MULT", ”RPAR"])      # expected token
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse a Unit?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |     ID
6: Op     ::= ‘+’
7:          |    ‘*’
```

How do we parse a Unit?

```
def parse_Unit(self):

    token_id = get_token_id(self.to_match)

    # Unit  ::= ‘(‘ Expr ‘)’
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return

    raise ParserException(-1,                  # line number (for you to do)
                          self.to_match,       # observed token
                          ["LPAR", "ID"])      # expected token
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse a Unit?

```
def parse_Unit(self):

    token_id = get_token_id(self.to_match)

    # Unit  ::= ‘(‘ Expr ‘)’
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return

    raise ParserException(-1,               # line number (for you to do)
                          self.to_match,    # observed token
                          ["LPAR", "ID"])   # expected token
```

ensure that to_match has token ID of “LPAREN” and get the next token

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:        | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:        |    ID
6: Op    ::= ‘+’
7:        |   ‘*’
```

How do we parse an Op?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |    ‘*’
```

How do we parse an Op?

```python
def parse_Op(self):

    token_id = get_token_id(self.to_match)

    # Op  ::= '+'
    if token_id == "PLUS":
        self.eat("PLUS")
        return

    # Op  ::= '*'
    if token_id == "MULT":
        self.eat("MULT")
        return


    raise ParserException(-1,             # line number (for you to do)
                          self.to_match,    # observed token
                          ["MULT", "PLUS"]) # expected token
```

First+ sets:

```
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ○ The number of non-terminals
- ○ The length of the input string
- ○ The number of tokens in the input string
- ○ How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Likely plays a small role, but typically the number of non-terminals is much smaller than the input string

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- The number of non-terminals
- The length of the input string
- The number of tokens in the input string
- How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Good answer, but potentially the input string is one giant ID. Then the parser simply needs to match one token.

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ○ The number of non-terminals
- ○ The length of the input string
- ○ The number of tokens in the input string
- ○ How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ○ The number of non-terminals
- ○ The length of the input string
- ○ The number of tokens in the input string
- ○ How many times a backtrack might occur

The parser needs to match every single token once. This is the correct answer

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Quiz

parsing An LL(1) grammar has a runtime proportional to:

- ◯ The number of non-terminals
- ◯ The length of the input string
- ◯ The number of tokens in the input string
- ◯ How many times a backtrack might occur

Backtracking is not required for LL(1) grammar

Review

Do we need backtracking?

The First+ set is the combination of First and Follow sets

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |    ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Do we need backtracking?

The First+ set is the combination of First and Follow sets

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We cannot select the next rule based on a single look ahead token!

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:        |   ID ‘[‘ Args ‘]’
3:        |   ID ‘(‘ Args ‘)’
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= ‘[‘ Args ‘]’
3:             |   ‘(‘ Args ‘)’
4:             |   “”
```

```
First
1: {ID}
2: {‘[‘}
3: {‘(‘}
4: {“”}
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:        |   ID '[' Args ']'
3:        |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:             |   '(' Args ')'
4:             |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

It is not always possible to rewrite grammars into a predictive form, but many programming languages can be.

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:              |   '(' Args ')'
4:              |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

New material

Scope

- What is scope?
- Can it be determined at compile time? Can it be determined at runtime?
- C vs. Python
- Anyone have any interesting scoping rules they know of?

Scope

- Lexical scope example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

What are the final values in x and y?

Scope

- We can catch certain variable scope errors at parse time
  - e.g. if a variable was declared in the current scope or not

Scope

- Lexical scope example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

This program should parse and execute

```
int x = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

What about this one?

Scope

- Lexical scope example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

This program should parse and execute

```
int x = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

undeclared!

What about this one?

How to track scope?

How to track scope?

- Symbol table object

- two methods:
  - **`lookup(id)` :** `lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

  - **`insert(id,info)` :** `insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

What information might we store about an id?

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
x++;
int y;
y++;
```

statements are either a declaration or an increment

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

statements are either a declaration or an increment

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

error!

statements are either a declaration or an increment

How to track scope?

- `SymbolTable ST;`

Say we are matched the statement:
`int x;`

declare_statement ::= TYPE ID SEMI

`{}`

`lookup(id)` `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

`insert(id,info)` `: insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

How to track scope?

- `SymbolTable ST;`

Say we are matched the statement:
`int x;`

declare_statement ::= TYPE ID SEMI

```
{
  self.eat(TYPE)
  variable_name = self.to_match[1] # lexeme value
  self.eat(ID)
  ST.insert(variable_name,None)
  self.eat(SEMI)
}
```

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

inc_statement ::= ID INCREMENT SEMI

```
{}
```

lookup(id) : lookup an id in the symbol table. Returns None if the id is not in the symbol table.

insert(id,info) : insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

inc_statement ::= ID INCREMENT SEMI

```
{
  variable_name = self.to_match[1] # lexeme value
  if ST.lookup(variable_name) is None:
      raise SymbolTableException(variable_name)
  self.eat(ID)
  self.eat(INCREMENT)
  self.eat(SEMI)
}
```

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

```
int x;
{
   int y;
   x++;
   y++;
}
y++;
```

start a new scope S

remove the scope S

How to track scope?

- Symbol table
- four methods:
  - **`lookup(id)`** `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`
  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`
  - **`push_scope()`** `: push a new scope to the symbol table`
  - **`pop_scope()`** `: pop a scope from the symbol table`

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

You will be adding the functions to push and pop scopes in your homework

How to implement a symbol table?

- Thoughts? What data structures are good at mapping strings?

- Symbol table
- four methods:
  - **`lookup(id)`** `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

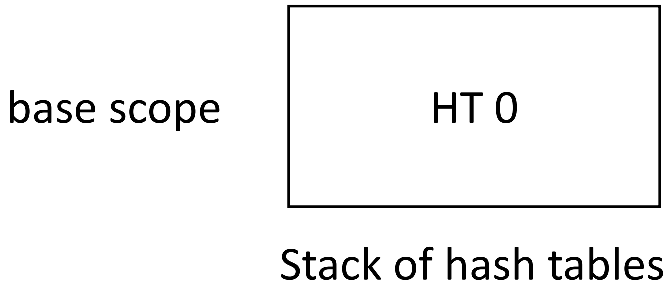
  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`

  - **`push_scope()`** `: push a new scope to the symbol table`

  - **`pop_scope()`** `: pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

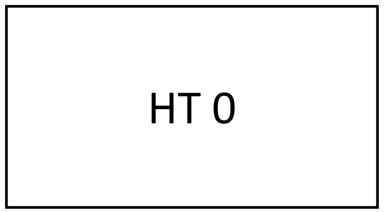


How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`push_scope()`



Stack of hash tables

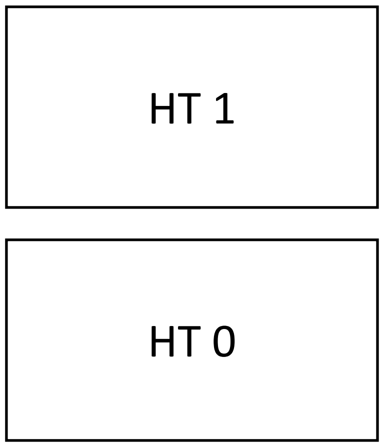
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

adds a new
Hash Table
to the top of the stack

`push_scope()`



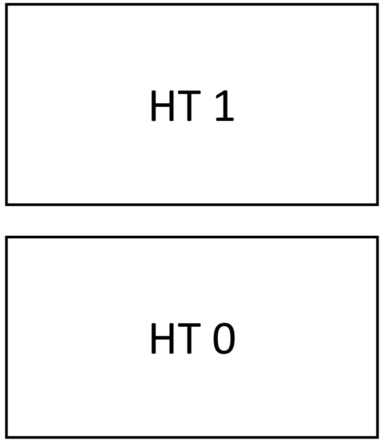
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



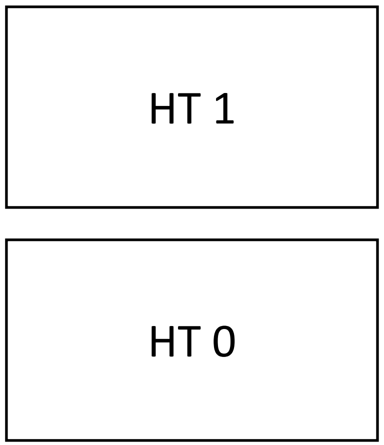
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

insert `(id -> data)` at top hash table

`insert(id,data)`



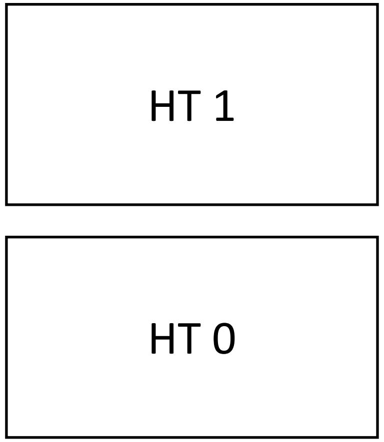
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

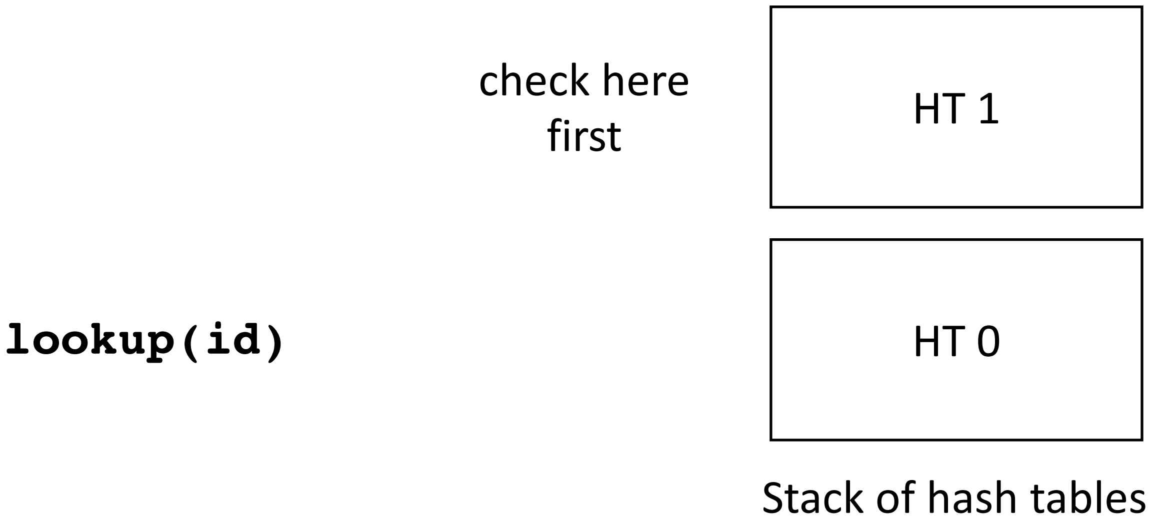


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



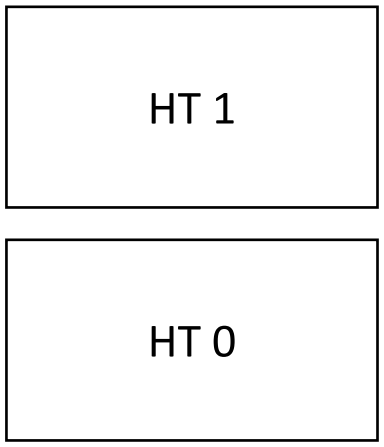
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

then check here

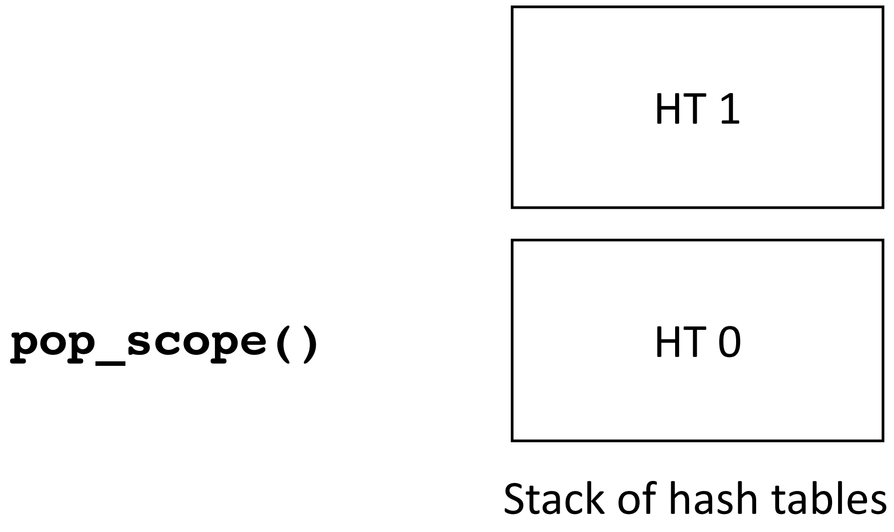


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

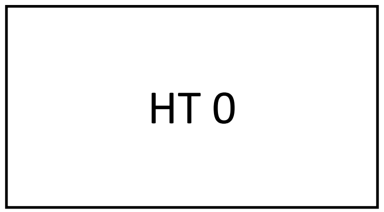
- A good way is a stack of hash tables:



How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



Stack of hash tables

How to implement a symbol table?

- Example

```
int x = 0;
{
  int y = 0;
  y++;
  x++;
}
x++;
y++;
```

HT 0

Stack of hash tables

Parser actions

Parser actions

- Like token actions: perform an action each time a production option is matched. Useful for: tracking state

Parser actions

- Like token actions: perform an action each time a production option is matched.

- Typically performed after the entire production action is matched

- Useful for:
  - tracking state

Example

- `SymbolTable ST;`

Say we are matched the statement:
`int x;`

declare_statement ::= TYPE ID SEMI

```
{
  self.eat(TYPE)
  variable_name = self.to_match[1] # lexeme value
  self.eat(ID)
  ST.insert(variable_name,None)
  self.eat(SEMI)
}
```

If we wrote our own recursive descent parser we can implement our own actions inlined

Example

Say we are matched the statement:
`int x;`

- `SymbolTable ST;`

Parser actions would be written like this

```
                       $1    $2    $3
declare_statement ::= TYPE  ID    SEMI
{
   ST.insert($2, None);
}
```

result of each symbol.
For a terminal it will be
the value

always some way to refer to symbol value, e.g. an array

What values get returned from non-terminals?

```
1: Expr   ::= Expr ‘+’ Unit      {print $1}
2:          |   Expr ‘-’ Unit
3:          |   Unit
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    NUM
```

What does this print?

What values get returned from non-terminals?

```
1: Expr   ::= Expr '+' Unit     {print $1; return "expr"}
2:          |   Expr '-' Unit   {return "expr"}
3:          |   Unit            {...}
4: Unit   ::= '(' Expr ')'
5:          |    NUM
```

Each production rule needs to return something

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr ‘+’ Unit    {}
2:         |   Expr ‘-’ Unit    {}
3:         |   Unit             {}
4: Unit  ::= ‘(‘ Expr ‘)’     {}
5:         |    NUM             {}
```

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr '+' Unit     {return $1 + $3}
2:        |   Expr '-' Unit    {return $1 - $3}
3:        |   Unit             {return $1}
4: Unit  ::= '(' Expr ')'      {return $2}
5:        |    NUM             {return $1}
```

Shortcomings of parser actions

Difficult to perform actions in the middle of a production

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

start a new scope S

remove the scope S

Parser generators

- You provide the CFG, along with some hints, you get a parser back
- They typically use bottom-up parsers
  - Algorithm is more complicated
  - Able to handle more types of grammars naturally
  - Able to naturally encode precedence and associativity
- Examples of tools:
  - Yacc, Antrl, PLY

calculator example

These slides follow the calculator example from the PLY documentation

calculator example

```
import ply.lex as lex

tokens = ["NUM", "MULT", "PLUS", "MINUS", "DIV", "LPAR", "RPAR"]

t_NUM = '[0-9]+'
t_MULT = '\*'
t_PLUS = '\+'
t_MINUS = '-'
t_DIV = '/'
t_LPAR = '\('
t_RPAR = ‘\)'

t_ignore = ' '

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    exit(1)

lexer = lex.lex()
```

Set up the lexer

calculator example

- *Import the library*

```
import ply.yacc as yacc
```

- Simple rule

```
def p_expr_num(p):
    "expr : NUM"
    p[0] = int(p[1])
```

functions are given prefixed by `p_`

production rules are the doc string

return values are stored in p[0]
children values are in p[1], p[2], etc.

calculator example

- *Try it out*

calculator example

- *Next rule*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]
```

- Try it again

calculator example

- Set associativity (and precedence)

```
precedence = (
    ('left', 'PLUS'),
)
```

calculator example

- *Next rules*

```
def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]


def p_expr_div(p):
    "expr : expr DIV expr"
    p[0] = p[1] / p[3]
```

```
precedence = [
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV'),
]
```

calculator example

- *Last rule for expressions*

```
def p_expr_par(p):
    "expr : LPAR expr RPAR"
    p[0] = p[2]
```

calculator example

- *An extra we can easily implement*

```
def p_expr_div(p):
    "expr : expr DIV expr"
    if p[3] == 0:
        print("divide by 0 error:")
        print("cannot divide: " + str(p[1]) + " by 0")
        exit(1)
    p[0] = p[1] / p[3]
```

calculator example

- *Combining rules:*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]


def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]
```

```
def p_expr_bin(p):
    """
    expr : expr PLUS expr
         | expr MINUS expr
         | expr MULT expr
    """
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    else:
        assert(False)
```

calculator example

- Other useful options
  - Error recovery
  - Error reporting (it is better in our top down parsers)

- Question: how would we do a calculator implementation in our C-simple grammar? It is not left recursive so it is not as natural...

See you on Wednesday!

- Work on HW 2
- Starting the next module: intermediate representations