CSE 3302
Programming Languages

Syntax (cont.)

Semantics

Chengkai Li
Fall 2007

Parsing

- Parsing:
  - Determine if a grammar can generate a given token string.
  - That is, to construct a parse tree for the token string.
- Two ways of constructing the parse tree
  - Top-down (from root towards leaves)
    - Can be constructed more easily by hand
  - Bottom-up (from leaves towards root)
    - Can handle a larger class of grammars
    - Parser generators tend to use bottom-up methods

Top-Down Parser

- Recursive-descent parser:
  - A special kind of top-down parser: single left-to-right scan, with one lookahead symbol.
  - Backtracking (trial-and-error) may happen
- Predictive parser:
  - The lookahead symbol determines which production to apply, without backtracking

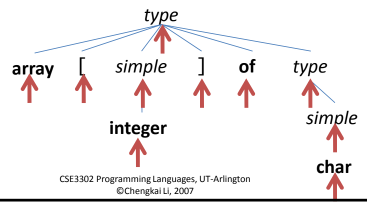
Recursive-Descent Parser

- Types in Pascal

type → *simple* | **array** [*simple*] **of** *type*
simple → **integer** | **char**

Input: **array** [**integer**] **of** **char**

Parse tree

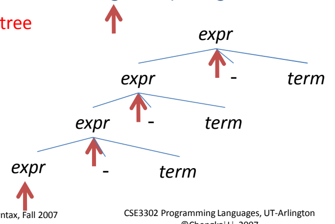


Challenge 1: Top-Down Parser Cannot Handle Left-Recursion

expr → *expr* - *term* | *term*
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Input: 3 – 4 – 5

Parse tree



Eliminating Left-Recursion

expr → *expr* - *term* | *term*
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⇒

expr → *term expr'*   *expr'* → – *term expr'* | ε
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(more complete example in Fig. 4.12, page 105-107)

```
void expr(void)
{ term();
  while (token == '-')
  { match('-');
    term();
  }
}
```

Challenge 2: Backtracking is Inefficient

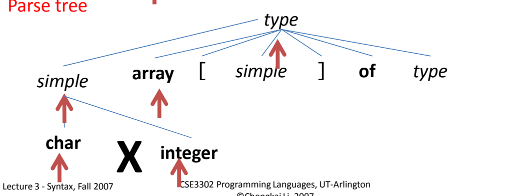
- Backtracking: trial-and-error

type → *simple* | **array** [*simple*] **of** *type* (Types in Pascal)

simple → **integer** | **char**

Input: **array** [**integer**] **of** **char**

Parse tree



Challenge 2: Backtracking is Inefficient

subscription → *term* | *term* .. *term*

term → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

- We cannot avoid backtracking if the grammar has multiple productions to apply, given a lookahead symbol.
- Solution: Predictive Parser
  - if we cannot decide early, defer the decision until we have seen enough.
  - Change the grammar so that there is only one applicable production that is unambiguously determined by lookahead symbol.

Avoiding Backtracking by Left Factoring

$A \rightarrow \alpha\ \beta1 \mid \alpha\ \beta2$

⇒

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta1 \mid \beta2$

expr → *term* | *term* .. *term*

term → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⇒

expr → *term rest*

rest → .. *term* | ε

term → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

EBNF

- Left-recursion removal and left factoring make grammar rules hard to write and read, and difficult to construct parser
- EBNF:

  Simplified notations, building predicative parser is simple

{ } repetition

expr → *expr* - *term* | *term*

⇒

expr → *term* {− *term*}

[ ] optional

if-stmt → `if` (*expr*) *stmt*

| `if` (*expr*) *stmt* `else` *stmt*

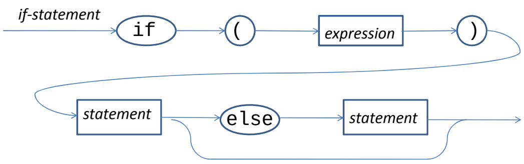
⇒

if-stmt → `if` (*expr*) *stmt* [`else` *stmt*]

Syntax Diagrams

- Written from EBNF, not BNF
- If-statement

  (more examples on page 101)



Review of Parsing

- Grammar for Integer Arithmetic Expression

expr → *expr* + *expr* | *expr* ∗ *expr* | (*expr*) | *number*

number → *number digit* | *digit*

digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

- Ambiguity 1: precedence

  (eliminated by precedence cascade)

expr → *expr* + *expr* | *term*

term → *term* ∗ *term* | (*expr*) | *number*

number → *number digit* | *digit*

digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Review of Parsing

- Ambiguity 2: Associativity (eliminated by recursion)

expr → *expr* + *term* | *term*
term → *term* ∗ *factor* | *factor*
factor → (*expr*) | *number*
number → *number digit* | *digit*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Ambiguity 3: Dangling-Else

Review of Parsing

- Parsers
  - top-down
    (construct the parse tree from root towards leaves)
    - recursive-descent
      (left-to-right scan, using single lookahead symbol)
      - predicative parser
        (no backtracking, the lookahead symbol unambiguously determines the production to apply)
  - bottom-up
    (construct the parse tree from leaves towards root)

Review of Parsing

- Allowing Top-Down Parsers : Eliminating Left-Recursion
- Allowing Predicative Parsers: Removing Left-Factoring
- Alternative representations: make predicative parser easy to write
  - EBNF
  - Syntax Diagram

Semantics

Names

- **Names**: identify language entities
  - variables, procedures, functions, constants, data types, …
- **Attributes**: properties of names
- Examples of attributes:
  - Data type:
    `int n = 5;` (data type: integer)
    - Note that `int` itself is a name.
  - Value: (value: 5)
  - Location:
    `int * y;`
    `y = new int;` (a new value for y, a new location for *y)
  - Parameters, return value: `int func(int n) {...}`
  - …

Binding Time

- Binding Time: the time when an attribute is bound to a name.
  - *Static binding* (static attribute):
    occurs before execution
    - *Language definition/implementation time*: The range of data type `int`
    - *translation time (parsing/semantic analysis)*: The data type of a variable
    - *link time*: The body of external function
    - *load time*: Location of global variable
  - *Dynamic binding* (dynamic attribute):
    occurs during execution
    - *entry/exit from procedure or program*: the value of local variables

Binding

- Binding: associating attributes to names
  - declarations
  - assignments
  - declarations (prototype) and definition of a function
- The bindings can be explicit or implicit
  `e.g. int x;`
  - Explicit binding: the data type of x
  - Implicit binding: the location of x (static or dynamic, depending on where the declaration is)
- The declaration itself can be implicit:
  - FORTRAN: no need to declare variables begin with I,J,K,L,M,N (integer variables)

Where can declarations happen?

- Blocks ({}, begin-end, … Algol descendants: C/C++, Java, Pascal, Ada, …)
  e.g., C
  - Function body
  - Anywhere a statement can appear (compound statement)
- External/global
- structured data type
- class

C++ Example

```
…
const int Maximum = 100;
struct FullName {string Lastname, string FirstName};

class Student {
private:
    struct FullName name; int Age;
public:
    void setValue(const int a, struct FullName name);
    int TStudent();
};

void Student::setAge(const int a, string lName, string fName) {
    int i;
    Age = a;
    {   int j;
        name.LastName = lName;
        name.FirstName = fName;
    }
}
```


Scope of Binding

- **Scope of Binding**: the region of the program where the binding is maintained (is valid, applies).
  (**Scope of Declaration**: if all relevant bindings by the declaration have identical scope.)
- **Block-structured language**

  ***lexical scope (static scope)*:** from the declaration to the end of the block containing the declaration.

  ***dynamic scope* :** introduced later.

Example

```
int x;
void p(void) {
   char y;
   . . .
   { int i;
     . . .          i    y
   }
}                                   p    x

void q(void) {
   double z;
   . . .          z
}                           q

main() {
   int w[10];
   . . .          w    main
}
```


Declaration before Use

```
void p(void) {
   int  x;
   . . .                x
   char y;     y
   . . .
}
```

Exception in OO languages: Scope of local declarations inside a class declaration includes the whole class.

```
public class {
   public int getValue() { return value; }     value
   int value;
}
```

Scope Hole

- Scope Hole: Declarations in nested blocks take precedence over the previous declarations. That is, binding becomes invisible/hidden.

```
int x;

void p(void) {
   char x;
   x = 'a';
   . . .
}

main() {
   x = 2;
   . . .
}
```

x (bound with character data type)

x (bound with integer data type)

Access Hidden Declarations

- scope resolution operator :: (C++)

```
int x;

void p(void) {
   char x;
   x = 'a';
   ::x=42;
   . . .
}

main() {
   x = 2;
   . . .
}
```

x (bound with character data type)

x (bound with integer data type)

the hidden integer variable x

Hide a Declaration

- File 1: extern int x; File 2: int x;

- File 1: extern int x; File 2: static int x;

Symbol Table

- Symbol Table: maintain bindings. Can be viewed as functions that map names to their attributes.

$$\text{Names} \xrightarrow{\text{SymbolTable}} \text{Attributes}$$

Symbol Table

```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

x: *integer*, *1*, global

y: *character*, *'a'*, global

Symbol Table

```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

The symbol table in **p**:
the bindings available in **p**

x: *double*, *2.5*, local to **p**; *integer*, *1*, global

y: *character*, *'a'*, global

Symbol Table

```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

The symbol table in **q**:
the bindings available in **q**

x: *integer, 1*, global

y: *integer, 42*, local to **q**; *character, 'a'*, global

Symbol Table

```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

The symbol table in **main**:
the bindings available in **main**

x: *character, 'b'*, local to **main**; *integer, 1*, global

y: *character, 'a'*, global

Symbol Table

- The symbol table in previous slides are built during compilation
- The bindings are used in generating the machine code
- Result:
  1
  a
- E.g., semantics of **q**

```
void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}
```

The symbol table in **q**:
the bindings available in **q**

x: *integer, 1*, global

y: *integer, 42*, local to **q**; *character, 'a'*, global

Static vs. Dynamic Scopes

- Static scope (lexical scope):
  – scope maintained statically (during compilation)
  – follow the layout of source codes
  – used in most languages
- Dynamic scope:
  – scope maintained dynamically (during execution)
  – follow the execution path
  – few languages use it (The bindings cannot be determined statically, may depend on user input).
    - Lisp: considered a bug by its inventor.
    - Perl: can choose lexical or dynamic scope

What if it used dynamic scope?

```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

x: *integer, 1*, global

y: *character, 'a'*, global

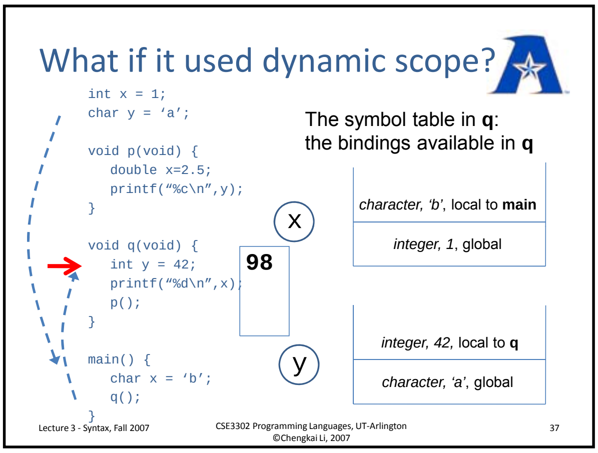
What if it used dynamic scope?

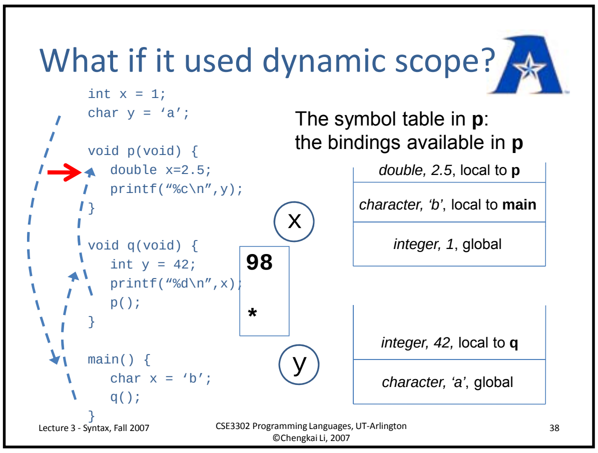
```
int x = 1;
char y = 'a';

void p(void) {
   double x=2.5;
   printf("%c\n",y);
}

void q(void) {
   int y = 42;
   printf("%d\n",x);
   p();
}

main() {
   char x = 'b';
   q();
}
```

The symbol table in **main**:
the bindings available in **main**

x: *character, 'b'*, local to **main**; *integer, 1*, global

y: *character, 'a'*, global


What if it used dynamic scope?
int x = 1;
char y = ‘a’;
void p(void) {
double x=2.5;
printf(“%c\n”,y);
}
void q(void) {
int y = 42;
printf(“%d\n”,x);
p();
}
main() {
char x = ‘b’;
q();
}
The symbol table in q:
the bindings available in q
x
y
98
character, ‘b’, local to main
integer, 1, global
integer, 42, local to q
character, ‘a’, global
Lecture 3 - Syntax, Fall 2007
CSE3302 Programming Languages, UT-Arlington
©Chengkai Li, 2007
37


What if it used dynamic scope?
int x = 1;
char y = ‘a’;
void p(void) {
double x=2.5;
printf(“%c\n”,y);
}
void q(void) {
int y = 42;
printf(“%d\n”,x);
p();
}
main() {
char x = ‘b’;
q();
}
The symbol table in p:
the bindings available in p
x
y
98
*
double, 2.5, local to p
character, ‘b’, local to main
integer, 1, global
integer, 42, local to q
character, ‘a’, global
Lecture 3 - Syntax, Fall 2007
CSE3302 Programming Languages, UT-Arlington
©Chengkai Li, 2007
38