Today's Topics

Last Week

- Project description
- S/SL language and implementation

This Week

- Scanners & Screeners
- Lexical & Syntactic Specification of Languages

Today

- Scanners & Screeners
- Hand out project phase 1: Qust Scanner/Screener

© 2020 J.R. Cordy

The Scanner

Lexical Analysis

- Phase 1 of the compiler (along with the Screener)
- Breaks up input text character stream into "tokens"
- A token is any sequence of characters to be treated as a unit, e.g., words, numbers, paired symbols such as <= and :=
- Exactly what is a token is a matter of contract between the Scanner/Screener and the Parser - must agree on representation

```
5.0E-3
```

- One token in the Turing compiler, but three tokens in the SP/k (PL/I) compiler

```
5.0   E   -3
```


© 2020 J.R. Cordy

The Scanner

Tokens

- Each token is represented in a compiler by an integer number (code)

Example Token	Token Name	Token Number
3, 120	pInteger	5
x, foo	pIdentifier	6
*	pStar	23
(	pLeftParen	17

- Tokens that have values associated with them (that is, whose text is different for each one, e.g., numbers, identifiers) are called *compound tokens* - their value (text) is emitted by the scanner along with their token

```
abc          pIdentifier   "abc"
27           pInteger      "27"
const        pIdentifier   "const"
```

The Scanner

Scanner Implementation

- Scanners may be implemented in three ways - *whole file* scanners, *single token* scanners, and *co-routine* scanners
- Whole file scanners are invoked as a separate program run that inputs the entire file of input text at once and outputs the entire file of tokens to an output file for input to a separate parser (rare)
- Single token scanners are functions called by the parser to return the next token in the input at a given point - they must reinitialize the scanner for each call - the only stored state is the current position in the input file (inefficient)

© 2020 J.R. Cordy

The Scanner

Scanner Implementation

- Co-routine scanners are subroutines called by the parser once to scan the entire file of input text, but they suspend and return each token as they recognize it
- This allows the parser to parse the token, run until it needs another token, then it suspend itself and allow the scanner to resume until it has another one, and so on
- The state of the scanner is preserved when the subroutine returns a token to the parser, and when the parser needs the next token, the scanner continues where it was suspended - this saves calling and reinitialization time for each token
- In effect the two are running in parallel, but sharing one CPU, which they pass back and forth like a baton
- Like multi-threading, but much more efficient

Scanners in S/SL

```
% Single token
Scan:
   [
      | '+': .pPlus
      | cLetter:
         @ScanIdentifier
     ...
      | cEof:
         .pEof

   ];
```

```
% Whole file / Co-routine
Scan:
   {[                              ←
      | '+': .pPlus
      | cLetter:
         @ScanIdentifier
     ...
      | cEof:
         .pEof
         >                         ←
   ]};                             ←
```

Example input:

```
x := x + 1
```


© 2020 J.R. Cordy

Character Classes

- Recall that S/SL choices are implemented as sequential lookups in a choice table

```
[
    | 'a','b','c',...,'z','A','B',...,'Z':
        @ScanIdentifier
    ...
]
```

```
    oInputChoice TBL

    TBL: 52
        'a' L1
        'b' L1
          ...
        'Z' L1
          ...
```

- Too inefficient for a scanner!

© 2020 J.R. Cordy

Character Classes

- Instead, all practical scanners prescan input characters using a low level input reader that characterizes each character according to its *character class* (i.e., letter, digit, etc.)

- This can be done efficiently using the input character's ASCII value as an index into a 256 element array (one for each possible ASCII character code)

```
a-z A-Z       cLetter
0-9           cDigit
+             cPlus
```

- Pseudo code for prescan:

```
read char c;
return charClass[c];
```

Character Classes

- Scanner S/SL choices are then made using the character class of the next input character rather than the actual character itself

```
[
    | cLetter:
       @ScanIdentifier
    | cDigit:
       @ScanNumber
    ...
]
```

```
oInputChoice TBL

TBL: 52
cLetter L1
cDigit L2
 ...
```

- Much faster!

© 2020 J.R. Cordy

The Buffer Mechanism

- Part of the Scanner's job is to accumulate the characters associated with compound tokens such as identifiers so that their text can be returned when the token is emitted
- The Buffer mechanism allows the Scanner to save characters in a text buffer that can be emitted with the token
- The buffer is controlled from the S/SL program using the semantic operation *oBufferSave*

```
ScanIdentifier:
    oBufferSave         % saves input character
    {[
        | cLetter, cDigit:
          oBufferSave
        | *: >
    ]}
    .pIdentifier;       % emits token with buffer text,
                        % and clears the buffer
```


© 2020 J.R. Cordy

The Screener

Distinguishing Compound Tokens

- The Screener is a filter between the Scanner and the Parser, that examines compound tokens (those with text), in order to:
- Distinguish keywords from identifiers for the parser
- Evaluate constants to their numeric value, for efficiency
- Make a table of identifiers, so that they may be represented as integer indexes in the table rather than as their text, for efficiency in manipulation and comparison
- Handle include files

© 2020 J.R. Cordy

Hash Functions

- Hash functions are used to speed up access to the Identifier Table
- A *hash* is a function mapping text of an identifier to an *almost* unique integer in the range of Identifier Table indexes
- In general, the hash of an identifier is not completely unique - when two identifiers map to the same index it is called a *collision*
- How do we know when a collision has occurred?

  The identifier table has an entry and its associated text is not the identifier we are searching for

- A *secondary hash* is used to find alternate indexes for the identifier
  - In PT, the first hash function, called the *primary hash*, is:

```
((firstchar << 7)+(lastchar << 4) + length) mod tblSize
```

  - The secondary hash in PT is:

```
((previous hash value) + length) mod tblSize
```

Hash Functions

Example

- E.G., if Identifier Table size = 1000, scanned identifier = “abc” (ASCII value of ‘a’ is 41, ASCII value of ‘c’ is 43)

```
PrimaryHash("abc")
      = (41 << 7 + 43 << 4 + 3) mod 1000
      = (41 * 128 + 43 * 16 + 3) mod 1000
      = (5248 + 688 + 3) mod 1000
      = 5939 mod 1000
      = 939
```

- If there is a different entry in the identifier table at location 939, say “axc”, we have a collision, and the next (secondary) hash value would be:

```
(939 + 3) mod 1000 = 942
```

- If there are more collisions (from “acc”, or from other identifiers that happen to hash to the same value as the secondary), then the next hash value would be 945, 948, etc. until we find the slot for “abc” or an empty slot in the table to put it in

The Screener as Semantic Mechanism

- The Screener is not *always* implemented as a separate filter between the Scanner and the Parser
- Sometimes it is integrated *directly* into the Scanner, and sometimes some of the functionality is moved to the Parser
- The *Turing* and *Concurrent Euclid* compilers both have identifier table handling and literal constant resolution implemented as semantic mechanisms directly in their S/SL Scanners

The "Ident" mechanism has operations:

```
oIdentLookup       % hash and find the ident
                   % whose text in Buffer
oIdentChoose >> OutputToken
                   % choose on the token for it
```

The "Value" mechanism has operations:

```
oValueEvaluate     % evaluate the number
                   % whose text in Buffer
oValueChoose >> ValueClass
                   % choose on int vs float
```


© 2020 J.R. Cordy

The Screener Include Mechanism

- *Include* directives in source are not seen by the Parser
- In some *Unix C* compilers, they are not even seen by the Scanner, but are implemented as a separate compiler pass (e.g., "cpp")
- In *Turing* and *Concurrent Euclid* they are handled by an *Include mechanism* in the combined S/SL Scanner/Screener
- The *Include* mechanism provides a stack of nested include files
- In these languages the **include** keyword must be followed by a string literal giving the file name to be included
- The new filename is pushed onto the stack and the file is opened
- Input characters are scanned from the new file until the end is reached (or a nested **include** directive is found)
- At the end, the file is closed, popped from the stack and scanning resumes in the previous file on top of the stack (following the point where the include directive was)

The Screener Include Mechanism

file 'm.e':

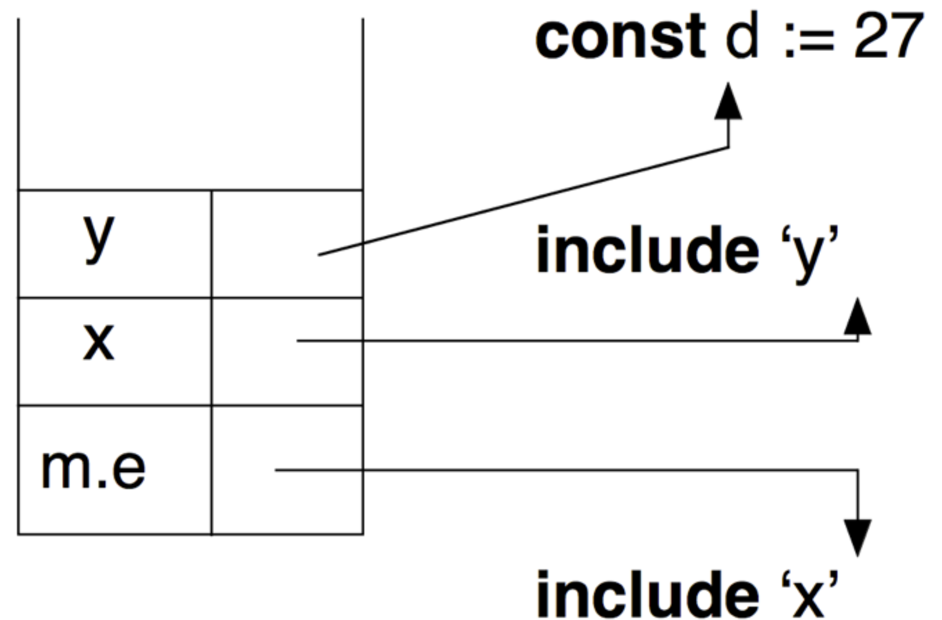
```
var m: module
    include 'x'
    ...
end module
```

file 'x':

```
procedure x
begin
    const c := 4
    include 'y'
    ...
end x
```

file 'y':

```
const d := 27
```



© 2020 J.R. Cordy

The Screener Include Mechanism

```
mechanism Include:
   oIncludePush
   oIncludePop
   oIncludeDepth >> number;

ScannerScreener:
   oIncludePush
   {[
      | cLetter:
         @Identifier
         oIdentLookup
         [oIdentChoose
            | pIdent:
               .pIdent
            | pInclude:
               @StringLiteral
               oIncludePush
            ...
         ]
      ...
      | cEndFile:
         [oIncludeDepth
            | one:
               >
            |*:
               oIncludePop
         ]
   ]}
   oIncludePop;
```

Summary

Scanner/Screeners

- Scanners break up input text into the tokens of the language
- Use character classes to efficiently classify input text characters
- Whole file, single token, and co-routine scanners
- Screeners filter recognized tokens to distinguish keywords, evaluate constants, tabulate identifiers
- May be filter, or directly integrated with Scanner using semantic mechanisms
- Screeners in S/SL, the Ident, Value and Include mechanisms

Next

- Lexical & Syntactic Specification of Languages