Today’s Topics

Previously

• Began looking at the S/SL computational model

This Time

• S/SL program structure and operations
• Begin with SL (Syntax Language), S/SL without semantic mechanisms
SL - S/SL Without Mechanisms

Syntax Language

• Without semantic mechanisms, SL can only recognize input tokens, push and pop the return stack and generate output and error tokens
SL - S/SL Without Mechanisms

Syntax Language

- Without semantic mechanisms, SL can only recognize input tokens, push and pop the return stack and generate output and error tokens.
- Mathematically equivalent to a push down transducer.
SL - S/SL Without Mechanisms

Syntax Language

• Without semantic mechanisms, SL can only recognize input tokens, push and pop the return stack and generate output and error tokens

• Mathematically equivalent to a push down transducer

• Therefore grammar class: context-free languages
S/SL Syntax

Chicken Scratchings

- S/SL is a very terse language, and looks a lot like “chicken scratchings” (until you get used to it)
- Most statements and operations are represented by single characters such as:

  
  : declaration
  { } loop statement
  > loop exit
  [ ] case/if statement
  | case/if alternative
  @ call statement
  >> return statement

- Comments use the % to end-of-line convention, like // in Java

  % This is a comment
S/SL Program Structure

**Program Structure**

- **S/SL programs** have two main sections: **definitions** and **rules**
- **Definitions** give the names of **tokens**, **types** and **constants** used in the program
- **Rules** are a set of **subprograms** defining the **actions** of the S/SL program
- Execution begins with the **first rule**
- We will look at **rules** first, then **definitions**
S/SL Program Structure

% Generic S/SL program

input:
    input token definitions;

output:
    output token definitions;

type Type:
    type constant definitions;

mechanism Mechanism:
    semantic mechanism operation definitions;

rules

FirstRule:
    actions;

OtherRules:
    actions;

end
S/SL Rules

Rules and Actions

• S/SL “rules” (subprograms) have one of two forms:

  name :  % procedure rule
            actions ;

  name >> type :  % “choice” rule (function)
            actions ;

• S/SL “actions” correspond to statements in other languages
S/SL Rules

Rule Call and Return

• Rules are called using the action @

• Rules return using the >> action, or by falling off the end of the rule

ProcedureDef:
   @ProcedureHeader
   @ProcedureBody;

DoStuff:
   @DoFirstThing
   >>
   @DoOtherThing;

Foo >> SymbolKind:
   @Bar
   >> sVar;

DoNothing:
   ;
SL Actions

**Actions**

- The SL subset of S/SL has 8 actions

- **Call** @ - call
- **Return** >> - return
- **Input** x - recognize an input token (implicit)
- **Emit** .x - generate an output token
- **Error** #x - generate an error token
- **Cycle** { } - repeat a sequence of actions (loop)
- **Exit** > - exit a cycle (loop)
- **Choice** [ ] - choose between sets of actions (if/case/switch)
SL Input Action

Input in S/SL

- The input action is implicit - there is no action symbol for it
- To require a particular token as input, we just write it as an action
- Means that the next token in the input must be the one named (i.e., we are “expecting” it as the next input token)
- The token may be specified in one of three forms:
  - a symbolic name (e.g. pColonEquals)
  - a string synonym (e.g. ‘:=’, the text of the token in quotes)
  - a wildcard that matches any next input token ( ? )

Assign:
  pIdentifier pColonEquals pInteger pSemicolon;

Assign:
  pIdentifier ‘:=’ pInteger ‘;’ ;

- If the next input token does not match, S/SL generates an error and syntax error recovery is invoked
SL Emit Action

Output in S/SL

• The emit (token output) action is indicated using a period character ( . ) followed by the token to be output

• The specified output token is emitted to the output stream

   Expr:
   @Term ‘+’ @Term .sAdd;

   Term:
   pInt .sIntLit ‘*’ pInt .sIntLit .sMult;

• Example:

   pInt(3) * pInt(5) + pInt(7) * pInt(8)
   →
   sIntLit(3) sIntLit(5) sMult sIntLit(7) sIntLit(8) sMult sAdd

• That is:

   3 * 5 + 7 * 8 → 3 5 * 7 8 * +
SL Error Action

Reporting Errors from S/SL

• The error (emit error token) action is indicated using a # followed by the token to be emitted

• Examples:

  #eMissingSemicolon
  #eTypeMismatch

• Emits the specified error token to the error output stream
**SL Cycle and Cycle Exit Actions**

**Loops in S/SL**

- Cycles specify repetition (looping)
  ```
  {  
    actions  
  }  
  ```
- Actions within the cycle are repeated until the cycle is exited (using an **exit** action `>` ) or the rule is exited (using a **return** action `>>` )
- Cycles may be nested, and the exit action terminates only the *innermost* cycle containing the exit
  ```
  {  
    {  
      >  % still an infinite loop  
    }  
  }  
  ```
SL Choice Action

Decisions in S/SL

• The `choice` action implements conditional flow of control, like an `if`, `case` or `switch` statement in other languages

```plaintext
[ selector
  | labels :
    actions
  | labels :
    actions
  ...

  | * :
    actions
]
```

• The optional `selector` can be a choice rule (function) call, or nothing

• If the `selector` is absent, the choice is made on the next input token in the input stream

• If no alternative label matches, S/SL generates an error and syntax error recovery is invoked
SL Actions - An Example

AssignOrCall:
  pIdentifier
  @OptionalSubscript
  [ 
    | ':=':
      @Expression
    | '*:'
  ] ';' 

OptionalSubscript:
  [ 
    | '(':
      @Expression ')'
    | '*:'
  ] ;

Expression:
  [ 
    | pIdentifier:
      @OptionalSubscript
    | pInteger: 
  ];

A(J) := 1;
P(6);
A(B(C));
B := 1;
X := J(6);
SL Actions - Another Example

SomeRule:
  [@CommaOrParenthesis
   | true:
     actions
   | false:
     actions
  ];

CommaOrParenthesis >> Boolean:
  [
    | ',',:
      >> true
    | ')':
      >> false
  ];
SL Actions - Another Example

SomeRule:
[@CommaOrParenthesis
  | true:
    actions
  | false:
    actions
];

CommaOrParenthesis >> Boolean:
[
  | ‘,’:
    actions
  | ‘)’:
    actions
];
S/SL Definitions

Declarations in S/SL

- Definitions in S/SL play the role of type and constant declarations in other languages.
- Specify the names of input, output, and error tokens as well as other user-defined types.

**input:**
- `pIdentifier`
- `pInteger`
- `pPlus` ‘+’
- `pColonEquals` ‘:=’
- `pSemicolon` ‘;’;

**output:**
- `sInteger`
- `sAdd`
- `sSubtract`;

**error:**
- `eMissingSemicolon`;
S/SL Type Definitions

Types in S/SL

• User-defined types are returned by choice rules and used to communicate with semantic mechanisms
• They specify an ordered set of named values, much like an enumerated type (“enum”) in C++

```c
type Boolean:
    true
    false;

type SymbolKind:
    syVariable
    syConstant
    syType
    syProcedure;
```
Implementation of S/SL Types

Host Language Representation

- The enumerated values are represented in the host language (in our case PT Pascal) as integer constants

<table>
<thead>
<tr>
<th>S/SL</th>
<th>PT Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>output:</td>
<td>const</td>
</tr>
<tr>
<td>sInteger</td>
<td>sInteger = 0;</td>
</tr>
<tr>
<td>sAdd</td>
<td>sAdd = 1;</td>
</tr>
<tr>
<td>sSubtract;</td>
<td>sSubtract = 2;</td>
</tr>
<tr>
<td>type SymbolKind:</td>
<td>const</td>
</tr>
<tr>
<td>syVariable</td>
<td>syVariable = 0;</td>
</tr>
<tr>
<td>syConstant</td>
<td>syConstant = 1;</td>
</tr>
<tr>
<td>syType</td>
<td>syType = 2;</td>
</tr>
<tr>
<td>syProcedure;</td>
<td>syProcedure = 3;</td>
</tr>
</tbody>
</table>
Controlling Values of S/SL Types

Host Language Representation

- We can explicitly control the values used to represent tokens by optionally providing a value to be used.
- This has no effect on the S/SL program, but constrains the implementation to encode the token using the given value (usually for external reasons).

<table>
<thead>
<tr>
<th>S/SL</th>
<th>PT Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>input:</td>
<td>const</td>
</tr>
<tr>
<td>pIdentifier = 7</td>
<td>pIdentifier = 7;</td>
</tr>
<tr>
<td>pInteger = 33</td>
<td>pInteger = 33;</td>
</tr>
<tr>
<td>pPlus ‘+’ = 24;</td>
<td>pPlus = 24;</td>
</tr>
<tr>
<td>type number:</td>
<td>const</td>
</tr>
<tr>
<td>zero = 0</td>
<td>zero = 0;</td>
</tr>
<tr>
<td>one = 1</td>
<td>one = 1;</td>
</tr>
<tr>
<td>two = 2</td>
<td>two = 2;</td>
</tr>
<tr>
<td>eight = 8;</td>
<td>eight = 8;</td>
</tr>
</tbody>
</table>
Summary

The S/SL Language

• Syntax based on single character actions
• SL subset equivalent to pushdown transducer (i.e., context-free parser)
• Input, output, cycle, choice, call, return actions used in subprograms called rules
• Only constant values, specified as enumerated types

Next

• Semantic mechanisms, whole S/SL programs, and implementation details