Proyecto 1
Analizador léxico y sintáctico para un subconjunto de Python

9 de marzo de 2010

Fecha de entrega: 4 de Abril de 2010

El presente proyecto tiene como objetivo realizar las dos primeras etapas en el proceso de compilación: el análisis léxico y el análisis sintáctico.

Por tanto, el programa correspondiente a estas dos etapas tiene que recibir como entrada, código fuente en Python y producir como salida una representación intermedia que corresponda a la entrada, existen diferentes representaciones intermedias, nosotros usaremos árboles de sintaxis abstracta (*AST*) por sus siglas en inglés.

Para realizar el análisis léxico, tienes dos opciones que realices la implementación a "mano" o que hagas uso de *flex*[1]. En cuanto al analizador sintáctico puedes hacer uso de *bison*[2], o escribir tu propio analizador sintáctico de descenso recursivo. Recuerda que una parte imporante de tu proyecto es que realices un buen conjunto de pruebas y que tu programa se comporte de manera correcta bajo ellas.

En principio trabajaremos con la especificación oficial de Python[3] en su versión 2.5. Sin embargo para faciliar nuestra labor por el momento no tomaremos en cuenta lo siguiente:

1. En cuanto a la estructura léxica
   - *a*) Cadenas con formato unicode, por ejemplo `u''hola''`.
   - *b*) Enteros largos. Todos los enteros deben estar dentro del rango $[-2^{31}, 2^{31}]$.
   - *c*) Números imaginarios.
2. En cuanto a expresiones
   - *a*) Listas por comprensión, ni expresiones generadoras.
   - *b*) Conversiones de cadenas.
   - *c*) trozos extendidos (*extended slices*).
   - *d*) Argumentos con palabras clave, argumentos `*`, o argumentos `**`, ni parámetros por omisión.
   - *e*) El operador obsoleto `<>`.
3. En cuanto a enunciados
   - *a*) El enunciado `del`.
   - *b*) El enunciado `yield` (utilizado en generadores y corrutinas).
   - *c*) Solamente se tomara en cuenta la sintaxis `raise` y `raise E` para generar excepciones.

[1] http://flex.sourceforge.net/
[2] http://www.gnu.org/software/bison/
[3] http://www.python.org/doc/2.5

d) Unicamente se tomaran en cuenta los enunciados `import` que tengan la sintaxis
`import lista_de_idenficadores_simples`
o que tegan la sintaxis
`from identificador import lista_de_identificadores_simples`.

e) El enunciado `future`.

f) El enunciado `exec`.

g) Los enunciados `finally` y `else` dentro de un enunciado `try`.

h) El enunciado `with`.

i) Unicamente se permitaran declaraciones de clases con una única clase padre (herencia simple).

j) Decoradores.

k) Las declaraciones de clases y los enunciados `import` no pueden estar anidados dentro de alguna otra expresión.

l) No se pueden declara funciones dentro de enunciados `if`, `while`, `for` o `try`.

m) El enunciado `assert`.

n) Cuando se declara una función ésta solo puede contener identificadores simples como parámetros, por ejemplo no se admiten tuplas como parámetros.

ñ) Un enunciado `global` para una variable V debe aparecer antes de cualquier asignación de V en una función o clase dada.

El nombre de tu programa debe ser `pythocomp` (Python compiler). La salida de tu programa `pythcomp` debe ser un archivo de texto con la extensión `.ast` por cada archivo de entrada con extensión `.py`, cada archivo `.ast` representa el árbol de sintaxis abstracta de cada archivo de entrada `.py`. Los archivos de salida `.ast` del presente proyecto, serán los archivos de entrada para el siguiente proyecto (el analizador semántico), de esta forma se llevará a cabo la comuniación entre las diferentes etapas del compilador, debemos notar que esta comunicación se puede dar de manera más directa y eficiente por ejemplo haciendo usos de una estructura de datos que represente el árbol de sintaxis abstracta en memoria, sin embargo por el momento es más adecuado hacer la comunicación mediante archivos porque hace más fácil la detección de errores en cada una de las etapas, además de que es un enfoque modular, donde son independientes cada una de las etapas del compilador.

Los árboles de sintaxis abstracta, se escribirán con una notación al estilo *Lisp*. Donde cada nodo del árbol se representa mediante: `(operador número_de_línea operando_1 ... operando_n)`

Por ejemplo para el siguiente archivo de entrada:

```
1.  #Programa de prueba.
2.  import defns
3.  def f(n):
4.      i=0
5.      while i <=n:
6.          if 1 < i %7 <=2:
7.              print i,
8.      else:
9.          s = i + 2; t+= s ** 2
10.         print ''s='', s, ''t ='', t
```

El correspondiente archivo de salida debe ser:

```
(module 0
   (import_module 2 (id 2 defns))
   (def 3 (id 3 f) (id_list 3 (id 3 n))
    (block 4
```

```
(assign 4 (id 4 i) (int_literal 4 0))
(while 5 (comparasion 5 (id 5 i) (id 5 <=) (id 5 n))
 (if 6 (comparasion 6 (int_literal 6 1)
                      (id 6 <)
                      (binop 6 (id 6 i) (id 6 %) (int_literal 6 7))
                      (id 6 <=)
                      (int_literal 6 2))
   (print 7 () (expr_list 7 (id 7 i)))
   ())
  (stmt_list 8
    (stmt_list 9
      (assign 9 (id 9 s) (binop 9 (id 9 i) (id 9 +) (int_literal 9 2)))
      (aug_assign 9 (id 9 t) (id 9 +)
                    (binop 9 (id 9 s) (id 9 **) (int_literal 9 2))))
    (println 10 () (expr_list 10
                    (string_literal 10 "s =")
                    (id 10 s)
                    (string_literal 10 "t =")
                    (id 10 t))))))))\}
```

El número de línea corresponde al número de línea del código fuente donde se encuentra el comienzo del enunciado que se esta traudciendo. Los operandos pueden ser nodos, cadenas entre comillas, literales enteros, símbolos, o el símbolo especial `()`, que denota un operando opcional que no esta presente. Hay enunciados que tienen diferentes formas de ser representados, por ejemplo si un `else` correspondiente a un `if` esta compuesto por un único enunciado, eres libre de representarlo como un *AST* para dicho enunciado, o como una lista de enunciados con un único enunciado dentro de ella. La traducción de las líneas 9 y 10 puede ser representada como una lista de enunciados (`stmt_list`) que contiene tres enunciados (en lugar de una lista de enunciados que contiene una lista de enunciados con dos elementos y un enunciado).

En general, el número de línea asociado con un enunciado es el número de línea donde comineza el enunciado en el código fuente. No tienes que preocuparte demasiado por este aspecto, pero el número de línea debe ser razonable.

Tu analizador sintáctico debe ser capaz de detectar y reportar errores sintácticos, mostrando errores de la sigueinte forma: `prueba.py:12: syntax error`

Y debe terminar con código de salida 1 si es que ocurrió uno o más errores. Tu programa debe ser capaz de recuperarse de los errores, descartando fragmentos de código erroneo e intentando continuar a partir de código bien formado tanto como sea posible, aunque en estos casos el árbol que se genere como salida es irrelevante.

La estrategia general es que el analizador léxico debe detectar átomos malformados, mientras el analizador sintáctico debe detectar combinaciones de átomos mal formadas. Algunos de los errores lexicos que se deben detectar son los siguientes:

- Cadenas con comillas simples que estan incompletas debido a un salto de línea.
- Cadenas con comillas triples que estan incompletas debido a un fin de archivo.
- Constantes enteras que son demasiado grandes.
- Caracteres que no deben ser interpretados como átomos.
- Cualquier uso de palabras reservadas que no usaremos dentro de nuestro subconjunto, no debemos permitir su uso como identificadores.
- Sangría inconsistente.

Los operadores que se usaran en los árboles de sintaxis abstracta se presetan en la siguiente figura en notación *BNF*. La gramática[4] usa los operadores * y + que tienen el mismo significado que en las expresiones regulares y parentesis sin comillas para agrupar.

```
Compilation: '(' "module" N Stmt* ')'

N: INT

Expr: '(' "binop" N Expr Id Expr ')'
    | '(' "comparasion" N Expr (Id Expr)+ ')'
    | '(' "unop" N Id Expr ')'
    | '(' "if_expr" N Expr Expr Expr ')'
    | '(' "and" N Expr Expr ')'
    | '(' "or" N Expr Expr ')'
    | '(' "lambda" N IdList Expr ')'
    | '(' "tuple" N Expr* ')'
    | '(' "list_display" N Expr* ')'
    | '(' "call" N Expr ExprList ')'
    | '(' "dict_display" N Pair* ')'
    | '(' "string_literal" N STRING ')'
    | '(' "int_literal" N INT ')'
    | '(' "float_literal" N FLOAT ')'
    | Target

Stmt : Expr
     | Assign
     | StmtList
     | '(' "aug_assign" N Target Id Expr ')'
     | '(' "print" N Expr0 ExprList ')'
     | '(' "println" N Expr0 ExprList ')'
     | '(' "return" N Expr0 ')'
     | '(' "raise" N Expr0 ')'
     | '(' "break" N ')'
     | '(' "continue" N ')'
     | '(' "import_module" N Id* ')'
     | '(' "import_from" N Id IdList ')'
     | '(' "global" N Id+ ')'
     | '(' "if" N Expr Stmt Stmt0 ')'
     | '(' "while" N Expr Stmt Stmt' ')'
     | '(' "for" N TargetList Expr Stmt Stmt0 ')'
     | '(' "try" N Stmt (Expr0 Target0 Stmt)+ ')'
     | '(' "def" N Id IdList Block ')'
     | '(' "class" N Id Id Block ')'

Assign : '(' "assign" N TargetList RightSide ')'

Block : '(' "block" N Stmt* ')'

ExprList : '(' "expr_list" N Expr* ')'

Expr0 : Expr | "()"
```

[4] http://www.python.org/doc/2.5/ref/grammar.txt

```
Id: '(' "id" N ID ')'

IdList : '(' "id_list" N Id* ')'

Pair: '(' "pair" N Expr Expr ')'

RightSide : Expr | Assign

StmtList : '(' "stmt_list" N Stmt* ')'

Stmt0 : Stmt | "()"

Target:
     Id
   | '(' "attributeref" N Expr Id ')'
   | '(' "subscription" N Expr Expr ')'
   | '(' "slicing" N Expr Expr0 Expr0 ')'

TargetList:
     Target
   | '(' "target_list" N TargetList+ ')'

Target0: Target | "()"
```

Además de los átomos que aparecen entre comillas simples en la gramática, se tienen los siguientes símbolos terminales:

- **INT** Denota un literal entero decimal no negativo.
- **STRING** Denota un literal de cadena encerrado entre comillas dobles. Estas literales deben usar secuencias de escape conformadas por un número octal de cuatro caracteres, así en lugar de comillas dobles escribimos `\042`, en lugar de diagonal invertida escribimos `\134`, y en lugar de toodos los caracteres con código ASCII menor que 32 escribimos `\000-\037`. Estas no deben contener cualquier otra secuencia de escape. Por ejemplo la cadena

  ```
  "C:\\directorio\040contenido\t\"Hola, mundo!\"\n"
  ```

  se debe escribir como:

  ```
  "C:\134directorio contenido\011\042Hola, mundo!\042\012"
  ```

- **ID** Un símbolo que aparece sin comillas. Para propósitos del *AST*, los símbolos pueden contener letras, dígitos, guiones bajos y cualquiera de los símbolos correspondientes a los operadores de Python (pero no debe ser así, estos no son identificadores legales en los programas).
- **FLOAT** Un literal de punto flotante al estilo $C++/Java$ (de tipo double).

Veamos ahora los detalles de algunos *AST*. Todas las traducciones deben quedar en claro, por eso veremos algunos casos que no son obvios. En las siguientes descripciones, si X es un enunciado en Python, X' denota el *AST* correspondiente a X.

- **pass** No hay un nodo explícito para `pass`. Puedes simplemente omitir todos los enunciados `pass`, alternativamente puedes usar una lista de enunciados vacía en el *AST*.

- **binop y unop** Estos tipos de nodos representan a los operadores binarios y unarios de Python. En ambos casos, el operando Id es el símbolo del operador, por ejemplo `(id 5 +)`.

- **comparasion** Las comparaciones en Python tienen una regla de evaluación especial. Una comparación completa da como resultado `True` o `False`, pero una expresión como la que sigue `x < y < z` no es equivalente a `(x < y) < z`. En lugar de esto, si `x < y` es verdadero, este valor lo toma `y` que posteriormente es comparado con `z`. Si `x < y` es falso, entonces la comparación completa es falsa y `z` no se evalúa. Por tanto necesitamos un operador especial para comparaciones. Una vez más, los operandos Id son los símbolos de los operadores (`<`, `>`, `<=`, `>=`, `==`, `!=`, `in`, `notin`, `is`, `isnot`). Por simplicidad, usaremos `comparasion` para comparaciones simples (como `x < y`), aún cuando se comportan como operadores binarios ordinarios.

- **tuple** Traduce `(`E_1`,...,`E_k`)`. También traduce los casos en los que esta permitido que los paréntesis sean omitidos. Por ejemplo en los enunciados

  ```
  x = 2, 4, 6
  for x in 2, 4, 6: ...
  return 2, 4, 6
  ```

  `2,4,6` debe ser traducido como si fuese `(2,4,6)`. Por otra parte, cuando una lista como ésta se utiliza como un simple enunciado, como en:

  ```
  f(x), f(y), f(z)
  ```

  puedes traducirla como una tupla o puedes traducirla como una lista de tres enunciados.

- **assign** Hay un problema técnico cuando analizamos sintácticamente asignaciones como la siguiente:

  ```
  x, y, z = E
  ```

  ya que `x, y, z` es una expresión por sí sola. Como resultado, representaciones obvias de la gramática en *bison* cuasarán conflictos (¿estoy creando una lista de objetivos (TargetList) o una lista de expresiones (ExprList)? en realidad no lo sé hasta que veo `'='`). Esto se puede resolver facilmente analizando sintácticamente el lado izquierdo de una asignación como si se tratese expresión y después verificar el *AST* resultante, con una función auxilar en *C/C++* que se asegure que en efecto se trata de una lista de objetivos válida.

- **try** El código

  `try:`
  S_0
  `except` E_1`,` V_1`:`
  S_1
  `except ...`
  `...`
  `except` E_n`,` V_n`:`
  S_n

  se traduce a `(try` N S'_0 E'_1 V'_1 `...` E'_n V'_n S'_n`)`.

- **class** El primer `Id` es el nombre de la clase, y el segundo es el nombre de clase padre.

- **attributeref** Traduce $E.I$.

- **subscription** Traduce $E_1[E_2]$. La sintaxis de Python permite que E_2 sea una lista de expresiones; sin embargo solo se trata de una abreviatura. Traduce $X[A, B, C]$ como si se tratara de $X[(A, B, C)]$.

Como mencionamos anteriormete tu compilador debe ser ejecutado mediante el comando `pythcomp` y este debe recibir como primer argumento el número de etapa que se ejecutará, así que para este proyecto el primer argumento debe ser `1`, un ejemplo de la ejecución del compilador es el siguiente: `proyone$./pythcomp 1 uno.py dos.py` Debe producir como salida los archivos `uno.ast dos.ast`

También debe soportar la opción `-o` al estilo de *gcc* Por ejemplo: `proyone$./pythcomp 1 uno.py -o salida`

Debe generar el archivo `salida` correspondiente al *AST* del archivo `uno.py`

Opcionalmente puedes hace que tu analizador léxico produzca como salida archivos intermedios (con extensión `.lex`) que generen el trabajo realizado por el analizador léxico y sirvan como archivos de entrada para el analizador sitáctico y este sea el generé los archivos `.ast`. Es importante que tu código este bien documentado. Adicionalmente puedes usar herramientas como *make* para que faciliten tu trabajo de compilación y de ejecución de pruebas. Recuerda que tu proyecto debe cumplir los lineamientos.

Buena Suerte!