Projet : Analyseurs syntaxique et sémantique pour un sous-langage d'Ada

Assembleur – Compilation, ENSIIE

Semestre 3, 2021–22

1 Informations pratiques

Ce projet est à effectuer en binômes. Dès qu'un binôme se sera constitué, il enverra un mail à `guillaume.burel@ensiie.fr` pour vérification.

Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**.

L'analyseur syntaxique demandé à la question 2 sera impérativement obtenu avec les outils lex/yacc [1] ou ocamllex/ocamlyacc [2]. Ceci implique donc que votre projet sera écrit au choix en C ou en OCaml.

Des fichiers d'entrée pour tester votre code seront disponibles dans une archive à l'adresse : `http://www.ensiie.fr/~guillaume.burel/compilation/projet_little_Ada.tar.gz` . Vous attacherez un soin particulier à ce que ces exemples fonctionnent. (Il y a à la fois des exemples d'entrées correctes et d'entrées que le compilateur est censé rejeter.)

Votre projet est à déposer sur `http://exam.ensiie.fr` dans le dépôt `asscom_proj_2021` sur forme d'une archive `tar.gz` **avant le 5 janvier 2022** inclus. **Tout projet rendu en retard se verra attribuer la note 0.** Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

Toute tentative de fraude (plagiat, etc.) sera sanctionnée. Si plusieurs projets ont **des sources trop similaires** (y compris sur une partie du code uniquement), *tous* leurs auteurs se verront attribuer la note 0/20.

2 Sujet

Ada est un langage de programmation impérative de la famille de Pascal, dont le nom rend hommage à Ada Lovelace qui est considérée comme le premier humain à avoir écrit un programme informatique.

Le but de ce projet est d'écrire un exécutable qui prend en entrée un programme écrit en Little Ada, un sous-langage d'Ada, qui construit son arbre de syntaxe abstraite et

1. Documentation disponible à la page `http://dinosaur.compilertools.net/`. On peut aussi utiliser les versions libres flex/bison.
2. " `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`

qui fait quelques analyses sémantiques dessus. La syntaxe de Little Ada est décrite par ce qui suit :

Un commentaire commence par `--` et se termine à la fin de la ligne.

Les identifiants en Little Ada commencent par une lettre (minuscule ou majuscule) qui est suivie d'un nombre potentiellement nul de lettres, de chiffres et d'underscore, avec la condition que deux underscores ne peuvent pas se suivre et que l'identifiant ne peut pas se terminer par un underscore. La casse des lettres n'est pas significative. Ainsi, `unidentifiant`, `UnIdentifiant` et `uNiDeNtIfIaNt` représentent le même identifiant. (Ce sera également le cas pour les différents mots-clés ci-après.)

Un identifiant qualifié est une suite non vide d'identifiants séparés par des points (ex : `un.iden_tifiant.qua_li_fie`).

Les constantes seront :

— des constantes décimales : une suite non vide de chiffres entre `0` et `9` et d'underscores, qui ne commence pas et ne finit pas par un underscore et qui ne contient pas deux underscores consécutifs ; suivie optionnellement par un point et une suite de même nature ; suivie optionnellement par un `e` (minuscule ou majuscule), un signe optionnel `+` ou `-` et une suite de même nature. Les underscores n'auront pas de signification : `1234567` est la même constante que `1_234_567` et que `1_2_3_4_5_6_7` ; (autres exemples : `12_3.4_56`, `12e-2_345`, `1.234_5E1_2`) ;

— des constantes avec base : une suite non vide de chiffres entre `0` et `9` et d'underscores, qui ne commence pas et ne finit pas par un underscore et qui ne contient pas deux underscores consécutifs, qui correspond à un entier qui est la base dans laquelle la constante est définie, et qui doit être compris entre 2 et 16 ; suivie d'un dièse `#` ; suivi d'une suite non vide de chiffres entre `0` et `9`, de lettres entre `a` et `f` (minuscule ou majuscule) et d'underscores, qui ne commence pas et ne finit pas par un underscore et qui ne contient pas deux underscores consécutifs, où la valeur de chaque chiffre est strictement inférieure à la base (avec `a` qui vaut 10, ..., `f` qui vaut 15) ; suivie optionnellement par un point et une suite de même nature ; suivie d'un dièse `#` ; suivi optionnellement par un `e` (minuscule ou majuscule), un signe `+` ou `-` optionel et une suite de non vide de chiffres entre `0` et `9` et d'underscores, qui ne commence pas et ne finit pas par un underscore et qui ne contient pas deux underscores consécutifs. La base et l'exposant sont donc en notation décimale ; tandis que le nombre entier ou à virgule entre les dièses est exprimé dans la base. Ainsi, b`#`$c_n c_{n-1} \ldots c_0$`.`$c_{-1} \ldots c_m$`#e`p est égal à $\left(\sum_{j=m}^{n} c_j b^j\right) b^p$; (exemples : `10#1_234_567#`, `2#1001.0110#E-2`, `12#A02.b5#e-3`, `16#Dead_Beef#`) ;

— des constantes de chaînes de caractères : entre deux `"`, une suite éventuellement vide dont chaque élément est soit `""` (deux guillemets consécutifs), soit un caractère différent d'un retour à la ligne ou de `"` ; les doubles guillemets correspondent en fait à un seul guillemet de la chaîne ; ainsi, `"ab""\b"` représente la chaîne qui contient les 5 caractères `a b " \` et `b` ; (autres exemples `"une chaine"`, `"une autre ""chaine"""`).

Une expression peut être :

— soit un identifiant (qualifié ou non) ; en fonction du contexte, cet identifiant représentera soit une variable, soit un appel de fonction sans paramètre ;

— soit une constante ;

— soit une expression précédée de `-` ou d'un des mots-clés `abs` et `not` ;

— soit deux expressions séparées par un des symboles ou mots-clés `+ - * / ** = /= <= >= < > mod rem and or xor` ; (`/=` représente l'inégalité) ;

— soit deux expressions séparées par les mots-clés `and then` (et coupe-circuit) ;

— soit deux expressions séparées par les mots-clés `or else` (ou coupe-circuit) ;

— soit un identifiant (qualifié ou non) suivi d'une parenthèse ouvrante, d'une liste non-vide d'expressions séparées par des virgules, et d'une parenthèse fermante ; en fonction du contexte, cela représentera soit une conversion de type (si l'identifiant est un type et s'il y a une seule expression entre les parenthèses), soit un appel de fonction ;

— soit une expression entre parenthèses.

Les opérateurs binaires `+ - * / mod rem` seront parenthésés implicitement à gauche, il faudra explicitement donner les parenthèses pour les autres en cas de besoin. On a les priorités suivantes (du plus prioritaire au moins prioritaire, les opérateurs sur la même ligne ont la même priorité) :

```
** not abs
* / mod rem
+ -
= /= <= >= < >
and or xor  and then  or else
```

Une instruction est une séquence potentiellement vide d'étiquettes constituées chacune d'un identifiant entre `<<` et `>>`, suivie de :

— soit l'instruction vide : le mot-clé `null` suivi d'un point-virgule ;

— soit une affectation : un identifiant (qualifié ou non) suivi de `:=`, d'une expression et d'un point-virgule ;

— soit un appel de procédure : un identifiant (qualifié ou non) suivi d'un point-virgule ; ou un identifiant (qualifié ou non) suivi d'une parenthèse ouvrante, d'une liste non-vide d'expressions séparées par des virgules, d'une parenthèse fermante et d'un point-virgule ; (exemples : `proc;`, `other(1+2, x);`) ;

— soit un boucle simple : un nom de boucle optionnel constitué d'un identifiant suivi de deux-points ; puis le mot-clé `loop`, une séquence non-vide d'instructions, les mots-clés `end loop` et un point-virgule ; optionnellement, le point-virgule peut être précédé du nom de la boucle ; (exemple : `maboucle: loop null; end loop maboucle;`) ;

— soit un boucle tant que : un nom de boucle optionnel constitué d'un identifiant suivi de deux-points ; puis le mot-clé `while` suivi d'une expression, le mot-clé `loop`, une séquence non-vide d'instructions, les mots-clés `end loop` et un point-virgule ; optionnellement, le point-virgule peut être précédé du nom de la boucle ; (exemple : `while x > 0 loop x := x / 2; end loop;`) ;

— soit un boucle pour tout : un nom de boucle optionnel constitué d'un identifiant suivi de deux-points ; puis le mot-clé `for` suivi d'un identifiant, du mot-clé `in`, de façon optionnelle du mot-clé `reverse`, et soit de deux expressions séparées par `..`, soit d'un type ; puis le mot-clé `loop`, une séquence non-vide d'instructions, les mots-clés `end loop` et un point-virgule ; optionnellement, le point-virgule peut être précédé du nom de la boucle ; (exemple :
`tantque: for x in 1..10 loop Put(x); end loop tantque;`) ;

— soit une conditionnelle : le mot-clé `if` ; une expression ; le mot-clé `then`, une séquence non-vide d'instructions ; zéro, une ou plusieurs fois le mot-clé `elsif` suivi d'une expression, du mot-clé `then` et d'une séquence non-vide d'instructions ; optionnellement le mot-clé `else` suivi d'une séquence non-vide d'instructions ; les mots-clés `end if` et un point-virgule ; (exemple : `if x = 0 then null; elsif x < 0 then null; elsif x > 0 then null; else Put("WTF?"); end if;`) ;

— soit une distinction de cas : le mot-clé `case`, une expression, le mot-clé `is`, une séquence non-vide d'alternatives, les mots-clés `end case` et un point-virgule ; chaque alternative est formée du mot-clé `when`, d'une séquence non-vide de choix séparés par des `|`, d'une flèche `=>` et d'une séquence non-vide d'instructions ; chaque choix est soit une expression, soit deux expressions séparées par `..`, soit le mot-clé `others` ; (exemple :

```
case x * 2 is
  0 | 2 => a := 1; b := x;
  3..10 => a := 0;
  others => b := 1;
end case;
```

) ;

— soit un saut : le mot-clé `goto`, un identifiant et un point-virgule ; l'identifiant se réfère à celui d'une étiquette entre `<< >>` ; (exemple : `<<self>> goto self;`) ;

— soit une instruction de sortie : le mot-clé `exit` ; optionnellement un identifiant correspondant à un nom de boucle ; optionnellement une condition formée du mot-clé `when` et d'une expression ; un point-virgule ; (exemples : `exit;`, `exit maboucle when x < 0;`) ;

— soit un retour de procédure : le mot-clé `return` et un point-virgule ;

— soit un retour de fonction : le mot-clé `return`, une expression et un point-virgule.

Un type est soit un identifiant (qualifié ou non) seul ; soit un identifiant (qualifié ou non), le mot-clé `range`, deux expressions séparées par `..` ; (exemples : `Boolean`, `Integer range 0..2**7`).

Une déclaration est soit :

— une déclaration d'objets : une liste non-vide d'identifiants séparés par des virgules ; deux-points `:` ; optionnellement le mot-clé `constant` ; optionnellement un type ; optionnellement un définition ; enfin un point-virgule ; une définition est formée de `:=` suivi d'une expression ; (exemples :

```
X, Y : Integer;
C : constant := 1;
zero : constant Integer range 0..1 := 0;
```

);

— une déclaration de type : le mot-clé `type`, un identifiant, les mots-clés `is range`, deux expressions séparées par `..`, un point-virgule ; (exemple : `type Ascii is range 0..2**7 - 1;`) ;

— une déclaration de sous-type : le mot-clé `subtype`, un identifiant, le mot-clé `is`, un type, un point-virgule ; (exemple : `subtype Signed_Char is Integer range -2**7..127;`) ;

— un renommage : une liste non-vide d'identifiants séparés par des virgules ; deux-points `:` ; un type ; le mot-clé `renames`, un identifiant qualifié, un point-virgule ; (exemple : `r : Boolean renames main.proc.z;`) ;

— une spécification de procédure : le mot-clé `procedure` ; un identifiant ; optionnellement entre parenthèse une séquence non-vide de paramètres séparés par des points-virgules ; un point-virgule ; chaque paramètre est une liste non-vide d'identifiants séparés par des virgules, deux-points `:`, un mode et un identifiant (qualifié ou non) correspondant à un nom de type ; le mode est soit rien, soit `in`, soit `in out`, soit `out` ; (exemples :

```
procedure proc1;
procedure proc2(a : Integer, b, c : in Boolean,
                d : out Integer, e, f : in out Boolean);
```

— une spécification de fonction : le mot-clé `function` ; un identifiant ; optionnellement entre parenthèse une séquence non-vide de paramètres séparés par des points-virgules ; le mot-clé `return` ; un identifiant (qualifié ou non) correspondant à un nom de type ; un point-virgule ; (exemple : `function f(a, b : in out Integer) return Boolean;`) ;

— une définition de procédure ou de fonction : comme pour une spécification, mais en ajoutant avant le point-virgule le mot-clé `is`, une séquence potentiellement vide de déclarations, le mot-clé `begin`, une séquence non-vide d'instructions, le mot-clé `end` et optionnellement le nom de la procédure/fonction (exemple :

```
function verbose_identity(x : Integer) return Integer is
  Put(x);
  return x;
end id;
```

).

Un fichier Little Ada contient **une** définition de procédure ou de fonction.

3 Questions

1. Définir des types de données correspondant à la syntaxe abstraite des programmes Little Ada (couvrant toute la grammaire). En particulier on définira entre autres des types `file`, `declaration`, `instruction` et `expression`.
2. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit un fichier Little Ada et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `file`).
3. Écrire une fonction `print_consts` qui prend en argument un AST de fichier Little Ada et qui affiche toutes les constantes apparaissant dans le programme, en revenant à la ligne entre chaque. On affichera ces constantes de façon usuelle (`%g` de `printf` pour les nombre en virgule flottante par exemple). Ainsi `12#A02.b5#e-3` sera affiché `0.835041`.
4. Écrire une fonction `check_affect` qui prend en argument un AST de fichier Little Ada et qui vérifie qu'on n'affecte pas de valeur dans les identifiants qui correspondent soit à des variables déclarées avec le mot-clé `constant` ou soit à des paramètres dont le mode ne contient pas `out`.
5. Écrire une fonction `check_scope` qui prend en argument un AST de fichier Little Ada et qui vérifie que les variables, les types, les fonctions et les procédures sont bien déclarées quand elles sont utilisées. Les identifiants qualifiés permettent d'accéder aux variables, types, etc. cachés par des redéfinitions dans des fonctions imbriquées. Les renommages permettent de créer des alias vers des variables existantes. Par exemple dans

```
1  procedure Main is
2     X : Integer;
3
4     procedure Child1 is
5        X : Integer := 1;
6     begin
7        null;
8     end Child1;
9
10    procedure Child2 is
11       X : Integer := 2;
12
13       procedure Child2_1 is
14         X : Integer := 21;
15         Y : Integer renames Main.Child2.X;
16       begin
17          Main.X := X + Main.Child2.X + Y;
18       end Child2_1;
19
20    begin
```

```
21        Child2_1;
22     end Child2;
23
24  begin
25     Child2;
26     Put(X);
27  end;
```

à la ligne 17, `Main.X` est la variable déclarée ligne 2, `X` celle déclarée ligne 14, `Main.Child2.X` celle déclarée ligne 11, et `Y` celle déclarée ligne 11 également. La variable déclarée ligne 5 n'est pas accessible depuis la ligne 17, qu'on utilise un identifiant qualifié ou non.

Les types `integer`, `boolean` et `float`, les constantes `true` et `false` ainsi que la procédure `Put` (qui prend un nombre variable d'argument) seront considérées comme prédéfinies et ne poseront donc pas de problème de portée.

6. Écrire un programme qui parse un fichier Little Ada, affiche ses constantes à l'aide de la fonction `print_consts` puis effectue les vérifications `check_affect` et `check_scope`, en renvoyant un code d'erreur différent de 0 au cas où ils ne passent pas.