Accueil

Professeur et Démonstrateurs

Horaires et locaux

Notes de cours

Démonstrations et devoirs

Examens

Liens utiles

Au sujet du livre de cours

Consultez vos notes

Foire aux questions (FAQ)

TP1: Vers un mini compilateur d'un mini Pascal (MiniPasc)

[Cahier des charges] | [Consignes] | [Détails pénibles]

Ce texte en version ps, en version pdf

Cadre général

Le but de ce travail est de vous faire programmer en langage C (ou C++) la réalisation d'un analyseur de codes sources programmés à l'aide d'un sous-ensemble du langage Pascal appelé **MiniPasc** dans la suite.

Un exécutable de ce qui vous est demandé de faire est installé sur le compte `dift2030`, vous pouvez l'utiliser pour comprendre, tester des parties peu claires. J'ai probablement laissé des bugs dans ce programme je le considère plus comme une aide que comme un exemple à suivre. Voici la commande que vous pouvez lancer pour analyser le fichier `source1.mp`:

```
cat /u/dift2030/Ressources/source1.mp |
 /u/dift2030/Ressources/tp1 -i -v
```

La grammaire de **MiniPasc** est décrite sous forme BNF comme suit (une version ascii de cette grammaire est disponible ici):

BNF de MiniPasc

```
programme -> program identificateur ; liste_declarations
 declaration_methodes instruction_composee


liste_declarations -> declaration
 {liste_declarations}
liste_declarations ->

declaration -> var declaration_corps ;

declaration_corps -> liste_identificateurs : type

liste_identificateurs -> identificateur { ,
 liste_identificateur }

type -> standard_type | array [ literal_entier ..
 literal_entier ] of standard_type
standard_type -> integer

declaration_methodes -> declaration_methode { ;
 declaration_methodes }
declaration_methodes ->
```

```
 declaration_methodes }
declaration_methodes ->

declaration_methode -> entete_methode
 liste_declarations instruction_composee

entete_methode -> procedure identificateur arguments ;

arguments -> ( liste_parametres )
arguments ->

liste_parametres -> declaration_corps { ;
 liste_parametres }

instruction_composee -> begin liste_instructions end

liste_instruction -> instruction { ; liste_instruction
 }
liste_instruction ->

instruction -> lvalue := expression |
               appel_methode  |
               instruction_composee  |
                if expression then instruction else
 instruction |
                while expression do instruction |
                write (liste_expressions) |
                read (liste_identificateurs)

lvalue -> identificateur { [expression]}

appel_methode -> identificateur ( liste_expression )

liste_expressions -> expression { , liste_expression }

expression -> facteur { addop facteur } | facteur {
 mulop facteur }

mulop -> * | /
addop -> + | -

facteur -> identificateur {[expression]} |
 literal_entier | ( expression )
```

Les éléments en vert sont les symboles du métalangage:

- `->` est le symbole de réécriture (souvent noté ::=)
- `{}` indique quelque chose d'optionnel; ex: `S -> A { B C }` est l'équivalent de:
  `S -> A`
  `S -> A B C`
- `|` indique le choix

Les éléments en caractères gras sont les mots réservés du langage (que votre analyseur lexical aura à charge de reconnaître); les éléments en rouge indiquent des unités (tokens) reconnues par l'analyseur lexical que vous devez écrire:

- `identificateur` est un nom d'au moins un caractère qui commence par une lettre et qui est suivi d'un nombre quelconque (éventuellement nul) de lettre ou de chiffres:
  `lettre (lettre|chiffre)*` avec `lettre = [a-zA-Z]` et `chiffre = [0-9]`
- `literal_entier` désigne une suite (non vide) de chiffres:
  `chiffre chiffre*`

Enfin une règle de la forme: `A ->`
indique que A peut dériver le vide (dit autrement, A peut ne pas être utilisé dans les

Enfin une règle de la forme: A ->
indique que A peut dériver le vide (dit autrement, A peut ne pas être utilisé dans les règles où il apparaît en partie droite).

Cahier des charges

La réalisation d'un compilateur est un travail incrémental, nous vous proposons de procéder dans l'ordre suivant (le détail de chaque étape suit):

- Étape 1 ([4 points]) - **Analyse lexicale**: réalisation d'un analyseur lexical de **MiniPasc**. C'est l'étape la plus simple (très peu de choses à changer par rapport à l'analyseur lexical vu en cours)
- Étape 2 ([8 points]) - **Analyse syntaxique**: réalisation d'un analyseur syntaxique de **MiniPasc** qui vérifie seulement si un programme source donné est bien conforme à la syntaxe précisée par la BNF. Cette étape ne nécessite aucun codage de la structure abstraite.
- Étape 3 ([3 points]) - **Analyse sémantique**: vérification de la cohérence sémantique du code (voir spécifications). Cette étape consiste essentiellement à compléter le code de l´étape 2 afin de mettre à jour une table de symboles décrite plus loin.

La suite de cette section vous donne les spécifications de chacune des étapes.

Étape 1 - Analyse lexicale

Votre analyseur a à charge d'éliminer les **commentaires** qui en **MiniPasc** commencent par une accolade ouvrante {, peuvent courir sur plusieurs lignes, et se terminent par une accolade fermante }. Il doit également être insensible à l'**indentation**. En particulier, le résultat de l'analyse lexicale doit être le même pour les fichiers sources source1.mp et source2.mp qui sont deux versions différemment indentées du même code source.

Il est possible que le code source à analyser soit incorrect du point de vue lexical, auquel cas votre analyseur doit l'indiquer à l'utilisateur et sortir proprement (pas de coredump). C'est par exemple le cas pour le code source source3.mp (un commentaire est ouvert mais pas fermé), ainsi que pour le code source4.mp (le symbole @ ne fait pas partie des symboles reconnus par **MiniPasc**).

À titre indicatif, voici la trace produite par un analyseur lexical qui reconnaît les lexèmes de **MiniPasc** pour les quatre fichiers sources mentionnés; les codes des tokens et leurs noms vous appartiennent, en revanche, les lexèmes retournés par cet analyseur sont ceux que votre analyseur devrait également retourner.

- source1.mp -> trace1
- source2.mp -> trace2
- source3.mp -> trace3
- source4.mp -> trace4

Étape 2 - Analyse syntaxique

Cette étape à pour but de vérifier qu'une suite de tokens (tels qu'identifiés par votre analyseur lexical), est bien conforme à la grammaire BNF de **MiniPasc**. Si tel n'est pas le cas, votre programme s'arrête à la première erreur rencontrée en indiquant le numéro de ligne dans le fichier source où le problème est rencontré (ou du moins détecté), suivi d'un court message spécifiant l'erreur.

À titre indicatif, voici la trace d'exécution de cette étape 2 pour un ensemble de fichiers:

À titre indicatif, voici la trace d'exécution de cette étape 2 pour un ensemble de fichiers:

Trace d'exécution

```
% cat source1.mp | tp1 -i -m Syn
SYNTAX: Program name tp1Source1 OK

% cat source2.mp | tp1 -i -m Syn
SYNTAX: Program name tp1Source2 OK

% cat source3.mp | tp1 -i -m Syn
---------------
ERROR (lexical) line: 71 EOF in comment
---------------

% cat source4.mp | tp1 -i -m Syn
---------------
ERROR (lexical) line: 16 lexeme inconnu
---------------

% cat source5.mp | tp1 -i
SYNTAX: Program name tp1Source5 OK
```

Voici d'autres sources qui possèdent tous des erreurs de syntaxe: [source6.mp], [source7.mp], [source8.mp], [source9.mp], [source10.mp]

Notez que l'étape 2 ne présuppose aucun traitement sémantique. Par exemple le source numéro 5 passe le test de syntaxe alors qu'il contient de nombreuses erreurs qui sont indiquées à même le code et expliquées dans la section qui suit.

Étape 3 - Vérification sémantique

Le but de cette étape est de détecter certaines incohérences sémantiques (c'est à dire de sens) dans le code source analysé. Les incohérences que votre programme doit rechercher sont décrites ci-après. Votre analyseur sémantique doit s'arrêter dès la première erreur rencontrée en indiquant à l'utilisateur le numéro de ligne où l'erreur a été détectée et la nature de l'erreur.

Voici la liste des erreurs sémantiques que votre programme doit détecter:

- Dans une déclaration de tableau (ex: `var tab: array [1..10] of integer;` qui déclare un tableau de 10 entiers indicés de 1 à 10); la borne inférieure du tableau doit être inférieure ou égale à sa borne supérieure (c'est une erreur commise dans le code source5.mp ligne 18).
- Une variable doit être déclarée avant son utilisation. Ce n'est par exemple pas le cas dans l'affectation `c := 4;` ligne 33 de source5.mp.
- Appel d'une procédure n'existant pas (ex: ligne 85 de source5.mp). L'identificateur employé peut cependant exister (c'est le cas en ligne 86 où `tab` est un identificateur -- de tableau -- qui existe).
  Note: ce problème est proche du précédant.
- déclaration multiple d'un identificateur (que ce soit un identificateur de variable, de tableau ou de procédure). C'est par exemple le cas en ligne 21 de source5.mp.
  Notez cependant que **MiniPasc** accepte qu'une variable locale masque une variable globale, comme c'est le cas en ligne 56 de source5.cmp. **MiniPasc** refuse cependant qu'une variable locale redéfinisse un paramètre formel (comme c'est le cas en ligne 57 de source5.mp).
- Utilisation d'une variable comme si c'était un tableau. C'est par exemple le cas de l'instruction ligne 34 de source5.mp car `y` n'est pas le nom d'un tableau mais d'un entier.
- Appel d'une procédure avec le mauvais nombre de paramètres (cas des lignes 91 et 92 de source5.mp)

- Appel d'une procédure avec le mauvais nombre de paramètres (cas des lignes 91 et 92 de source5.mp)
- Mauvaise indicage d'un tableau. C'est par exemple le cas dans les lignes 73 et 75 de source5.mp où l'indice du tableau n'est pas dans les bornes valides spécifiées à la déclaration.
  Notez que ce problème peut survenir aussi bien en partie gauche d'une affectation, qu'en partie droite. C'est l'erreur la plus difficile à trouver de toutes celles que l'on vous demande de détecter. Prenez par exemple l'instruction suivante (légitime en **MiniPasc**):
  `tab2[2+13] := 3;`
  Vérifier que `2+13` est un bon indice nécessite le calcul de l'expression; c'est également la même situation dans cet exemple:
  `tab2[((2-(3))*(2-3))] := 3;`
  En revanche, dans l'instruction suivante, il n'est pas possible *a priori* de détecter la valeur de l'indice car une variable intervient dans le calcul de l'expression:
  `tab2[((x+(3))+(2*3))] := 3;`
  Bref, considérez la détection de cette erreur comme un bonus qui n'est pas imposé.

Il existe d'autres types d'incohérence qu'il faudrait gérer pour réaliser un véritable compilateur, il ne vous est pas demander de les détecter (rien ne vous empêche cependant d'y travailler). Exemple de vérification envisageable mais absolument pas demandée: vérification de la correspondance des types entre les paramètres d'appels d'une méthode et les paramètres formels de cette méthode. **Attention** Vous pourriez être tenté(e) d'écrire un analyseur qui recherche ces erreurs de manière ad-hoc avec un succès possible pour certaines erreurs vu que **MiniPasc** est un langage très simple.

Par exemple vous pourriez à l'aide de `grep` (ou tout autre reconnaisseur d'expressions régulières) extraire les bornes d'un tableau puis vérifier qu'elles sont dans le bon ordre. Mais **ce qui vous est demandé ici** est de gérer une **table de symboles** à l'aide de laquelle vous allez pouvoir détecter au moment de l'analyse syntaxique les incohérences. En d'autres termes, il vous est demandé d'enrichir le code de l'étape 2 pour effectuer ce contrôle et non d'en créer un nouveau.

Table des symboles

Une table de symboles est une structure dans laquelle sont rangés tous les identificateurs rencontrés lors de l'analyse syntaxique. Chaque identificateur est accompagné de sa nature (ici variable, tableau ou procedure), ainsi que d'informations spécifiques (borne inférieure et supérieure pour un tableau, nombre de paramètres formels pour une procédure).

D'autres informations sont normalement stockées dans la table de symboles, comme par exemple le type de chaque identificateur (integer, float, etc.), mais il ne vous est pas demandé dans ce travail de faire de la vérificaion de type.

Le plus gros du travail de l'étape 3 consiste à alimenter cette table avec les identificateurs rencontrés dans les déclarations et à vérifier lorsqu'un identificateur est utilisé dans une instruction si cela est fait de manière cohérente.

Plus à ce sujet un peu plus tard (commencez déjà par coder les deux premières étapes).

Consignes

Une partie de la correction sera automatisée (la compilation ainsi que l'exécution de votre programme sur un ensemble de codes sources), aussi devez-vous respecter

Une partie de la correction sera automatisée (la compilation ainsi que l'exécution de votre programme sur un ensemble de codes sources), aussi devez-vous respecter les consignes qui suivent.

L'interface de votre programme

Vous devez remettre un seul programme exécutable **tp1** qui accomplit toutes les étapes que vous avez codées. Voici le synopsis (descriptif) que doit suivre votre programme. Tester l'exécutable mis à votre disposition pour plus de détails.

Synopsis de votre programme

```
tp1 [Options] source

Options (les informations entre crocher sont les
 options par défaut):

-m mod fonctionne dans le mode specifié [comp]
-v     be verbose [don't]
-i     lit le code source depuis stdin [lit depuis le
 fichier nommé]
-h     guess !

avec com dans [lex,syn, sem]
     lex: lance seulement l'analyse lexicale (etape 1)
     syn: lance l'analyse lexicale + syntaxique (etape
 2)
     sem: lance les analyses lexicale + syntaxique +
 semantique (etape 3)

Note: Lorsque l'option -i est utilisée  n'est pas
 requis

Ex:

cat source1.mp | tp1 -i -m lex  lance l'analyse
 lexicale seulement sur source1.mp
tp1 -m lex source1.mp           idem

cat source1.mp | tp1 -i -m syn  lance l'étape 2 sur
 source1.mp
tp1 -i -m syn source1.mp        idem

cat source1.mp | tp1 -i -m sem  lance l'etape 3 sur
 source1.mp
tp1 -i -m sem source1.mp        idem
```

Notez qu'un exemple de code permettant de gérer proprement la ligne de commande vous a été remis dans un solutionnaire.

Codes d'erreur

Lorsque votre programme détecte une erreur (lexicale, syntaxique ou sémantique), il doit s'arrêter après avoir affiché un message adéquat. Vous devez sortir à l'aide de l'appel `exit(x)` où `x` est un entier qui vaut -2 en cas d'erreur lexicale (étape 1), -4 en cas d'erreur syntaxique (étape 2) et -8 en cas d'erreur sémantique (étape 3). Lorsque le code est correctement analysé, votre programme doit retourner 0 (return 0; en dernière instruction de la méthode `main`).

Compilation

Votre travail sera compilé sur les machines linux du département. Il vous appartient de vérifier que votre code fonctionne sur cet environnement. Vous devez fournir un fichier **Makefile** qui sera utilisé pour compiler votre travail (cela facilitera également votre travail). Pour simplifier à l'extrême, prenons un exemple (vos

de vérifier que votre code fonctionne sur cet environnement. Vous devez fournir un fichier **Makefile** qui sera utilisé pour compiler votre travail (cela facilitera également votre travail). Pour simplifier à l'extrême, prenons un exemple (vos démonstrateurs vous en diront plus sur les makefiles en démonstration):

J'ai réalisé ce TP à l'aide des 4 fichiers C++ suivants: `lex.cc`, `syn.cc`, `sem.cc`,`tp1.cc`; la méthode `main` étant dans `tp1.cc`. Je peux compiler mes souces en tapant à chaque fois la commande:

```
g++ -o tp1 tp1.cc lex.cc sem.cc syn.cc
```

Je peux également mettre cette ligne dans un fichier de nom `Makefile` et taper à la place la commande: `make` (nous utiliserons `make` pour compiler votre programme).

Travail en binôme

Vous devez travailler en groupe de deux. **Le nom et le login de chaque personne doivent être reportés en entête de chaque source remis !**.

Vous devez remettre un seul travail par binôme (vérifiez que le nom et le login des deux personnes apparaît dans toutes les entêtes de vos codes). N'oubliez pas de remettre également le fichier `Makefile`.

Procédure de remise, date limite, barême

La commande de remise est la suivante (prenez garde de remettre sous pift2030 et pas dift2030):

```
remise pift2030 TP1 mesSources Makefile
```

Pour ceux qui ne sont pas familiés avec la commande remise, tapez `man remise` dans un terminal.

Vous avez jusqu'à lundi 21 octobre 2002 midi pour remettre votre travail. Le barême est en partie indiqué dans le sujet. Vous êtes notés sur 15 points, 4 points sont donnés (étape 1), 8 points sont très accessibles (étape 2); des 3 points restants (étape 3), certains sont faciles à prendre (ex: vérifier que les bornes d'un tableau sont dans le bon ordre).

N'oubliez pas que la notation dépend toujours de critères comme: la qualité (pertinence) du code, son organisation (structuration), sa lisibilité (commentaires, indentation, etc.), sa justesse (le code est-il conforme aux consignes ?). De plus, des consignes vous sont spécifiées, le fait de ne pas les suivre est forcément pénalisant.

Prenez le temps de lire ce sujet, et n'oubliez pas que vos démonstrateurs et moi-même sommes là pour toutes vos questions (évitez cependant avec moi celles du genre: "euh, c'est trop long le sujet, vous ne pouvez pas me faire un résumé ?").

[top]

dift2030@iro.umontreal.ca